Titre: Title:	Estimation par analyse statique de la bande-passante d'accélérateurs en synthèse de haut niveau sur FPGA
Auteur: Author:	Frédéric Fortier
Date:	2018
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Fortier, F. (2018). Estimation par analyse statique de la bande-passante d'accélérateurs en synthèse de haut niveau sur FPGA [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/3095/

UNIVERSITÉ DE MONTRÉAL

ESTIMATION PAR ANALYSE STATIQUE DE LA BANDE-PASSANTE D'ACCÉLÉRATEURS EN SYNTHÈSE DE HAUT NIVEAU SUR FPGA

FRÉDÉRIC FORTIER
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
MAI 2018

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ESTIMATION PAR ANALYSE STATIQUE DE LA BANDE-PASSANTE D'ACCÉLÉRATEURS EN SYNTHÈSE DE HAUT NIVEAU SUR FPGA

présenté par : FORTIER Frédéric
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

Mme NICOLESCU Gabriela, Doctorat, présidente
M. BOIS Guy, Ph. D., membre et directeur de recherche
M. LANGLOIS J.M. Pierre, Ph. D., membre et codirecteur de recherche
M. SAVARIA Yvon, Ph. D., membre

DÉDICACE

À Erica, pour son support moral tout le long du projet et pour avoir toléré mon horaire particulier et toutes ces longues soirées de rédaction.

Merci !

REMERCIEMENTS

J'aimerais d'abord remercier Guy et Pierre pour leur support et conseils tout le long de ce travail. Ceux-ci ont été essentiels à son succès, tant pour bien orienter ma recherche que pour bien en présenter les résultats.

J'aimerais aussi remercier l'équipe de Space Codesign pour leurs conseils, idées et pour toutes les discussions, techniques ou non, que nous avons eu. La réalisation de ce travail aurait été beaucoup plus difficile sans leur accueil chaleureux dans leurs locaux.

Finalement, j'aimerais remercier mes parents pour leur support tout au long de mes études et mes colocs des deux dernières années pour avoir rendue ma vie moins monotone.

RÉSUMÉ

L'accélération par coprocesseur sur FPGA de portions d'algorithmes logiciels exécutés sur un CPU à usage général est une solution utilisée depuis longtemps dans de nombreux systèmes embarqués lorsque le calcul à effectuer est trop complexe ou la quantité de données à traiter trop grande pour être réalisée par ce processeur trop général pour les contraintes de performance et de puissance données. Avec la fin de la loi de Moore, c'est également une option de plus en plus utilisée dans les centres de données pour pallier à la croissance exponentielle de la consommation de courant des approches CPU et GPGPU.

De plus, la réalisation de ces coprocesseurs, bien que restant une tâche plus complexe que la simple programmation d'un processeur, est énormément facilitée par la démocratisation des logiciels de synthèse de haut niveau (HLS), qui permettent la transformation automatisée de code écrit en langages logiciels (généralement un sous-ensemble statique du C/C++) vers des langages de description matérielle synthétisables (VHDL/Verilog). Bien qu'il soit souvent nécessaire d'apporter des modifications au code source pour obtenir de bons résultats, les outils de synthèse de haut niveau comportent généralement un estimateur de performance rapide de la micro-architecture développée, ce qui facilite un flot de développement itératif.

Cependant, en pratique, le potentiel de parallélisme et de concurrence des accélérateurs sur FPGA est souvent limité par la bande-passante vers la mémoire contenant les données à traiter ou par la latence des communications entre l'accélérateur et le processeur général qui le contrôle. De plus, l'estimation de cette bande-passante est un problème plus complexe qu'il ne paraît du premier coup d'œil, dépendant notamment de la taille et de la séquentialité des accès, du nombre d'accès simultanés, de la fréquence des différentes composantes du système, etc. Cette bande-passante varie également d'une configuration de contrôleur mémoire à une autre et le tout se complexifie avec les FPGA-SoC (SoC incluant processeurs physiques et partie logique programmable), qui comportent plusieurs chemins des données fixes différents vers leur partie FPGA. Finalement, dans la majorité des cas, la bande-passante atteignable est plus faible que le maximum théorique fourni avec la documentation du fabricant.

Cette problématique fait en sorte que bien que les outils existants permettent d'estimer facilement la performance du coprocesseur isolé, cette estimation ne peut être fiable sans considérer comment il est connecté au système mémoire. Les seuls moyens d'avoir des métriques de performance fiables sont donc la simulation ou la synthèse et exécution du système complet. Cependant, alors que l'estimation de performance du coprocesseur isolé ne prend que quelques secondes, la simulation ou la synthèse augmente ce délai à quelques dizaines de

minutes, ce qui augmente le temps de mise en marché ou mène à l'utilisation de solutions sous-optimales faute de temps de développement.

Il ressort des points précédents la nécessité d'un outil offrant, pour une suite de transactions mémoire donnée, la bande-passante atteignable sur divers modèles de FPGA (et chemins des données à l'intérieur du même FPGA-SoC) afin de déterminer rapidement si la bande-passante disponible sur un modèle ou un autre permet d'atteindre les contraintes de performance requises. Ceci permettrait à l'utilisateur d'un tel outil de faire rapidement certains choix architecturaux majeurs et de déterminer si le goulot d'étranglement du système (et donc l'endroit où les efforts d'amélioration devraient être mis) se trouve dans sa hiérarchie de mémoire. De plus, le but d'un tel outil étant de réduire le temps de développement, il est important d'automatiser le processus de détermination des caractéristiques des accès mémoire effectués par le coprocesseur.

Nous avons donc identifié, à l'aide d'un mélange d'expérimentation et de littérature préexistante, les différentes variables affectant habituellement la performance des divers chemins des données entre la mémoire externe et un coprocesseur sur FPGA ou FPGA-SoC, puis développé une méthodologie permettant de caractériser cette performance. Nous avons ensuite testé cette méthodologie sur un modèle spécifique de Zynq-7020 de Xilinx.

Nous avons ensuite développé, en utilisant les bibliothèques pour compilateur LLVM et le transpilateur Clang, un outil déterminant de manière statique les accès mémoires faits par un algorithme C/C++ destiné à synthétisé par un outil HLS.

Finalement, nous avons comparé les résultats de la performance estimée en combinant les accès mémoire obtenus par analyse statique et la caractérisation du système mémoire du Zynq-7020 à l'exécution du système réel pour une série de tests. Ceux-ci montrent que l'estimation, qui ne prend que quelques dizaines de secondes à exécuter, a une erreur moyenne de 9% et une erreur maximale de 25%.

ABSTRACT

FPGA acceleration of portions of code otherwise executed on a general purpose processor is a well known and frequently used solution for speeding up the execution of complex and data-heavy algorithms. This has been the case for around two decades in embedded systems, where power constraints limit the usefulness of inefficient general purpose solutions. However, with the end of Dennard scaling and Moore's law, FPGA acceleration is also increasingly used in datacenters, where traditional CPU and GPGPU approaches are limited by the always increasing current consumption required by many modern applications such as big data and machine learning.

Furthermore, the design of FPGA coprocessors, while still more complex than writing software, is facilitated by the recent democratization of High-Level Synthesis (HLS) tools, which allow the automated translation of high-level software to a hardware description (VHDL/Verilog) equivalent. While it is still generally necessary to modify the high-level code in order to produce good results, HLS tools usually ship with a fast performance estimator of the resulting micro-architecture, allowing for fast iterative development methodologies.

However, while FPGAs have great potential for parallelism and concurrence, in practice they are often limited by memory bandwidth and/or by the communications latency between the coprocessor and the general purpose CPU controlling it. In addition, estimating this memory bandwidth is much more complex than it can appear at first glance, since it depends on the size of the data transfer, the order of the accesses, the number of simultaneous accesses to memory, the width of the accessed data, the clock speed of both the FPGA and the memory, etc. This bandwidth also differs from one memory controller configuration to the other, and then everything is made more complex when SoC-FPGAs (SoCs including a hard processor and programmable logic) come into play, since they contain multiple different datapaths between the programmable logic and the hard memory controller. Finally, this bandwidth is almost always different (and smaller) than the maximum theoretical bandwidth given by the manufacturer's documentation.

Thus, while existing HLS tools can easily estimate the coprocessor's performance if it is isolated from the rest of the system, they do not take into account how this performance is affected by the achievable memory bandwidth. This makes the simulation of the whole system or its synthesis-then-execution the only trustworthy ways to get a good performance estimation. However, while the HLS tool's performance estimation runtime is a matter of a few seconds, simulation or synthesis takes tens of minutes, which considerably slows down

iterative development flows. This increased delay increases time-to-market and can lead to suboptimal solutions due to the extra development time needed.

These problems make a case for the necessity of a tool that can give, for a given memory transactions list, the associated communications latency on different FPGA models (and different datapaths within a given SoC-FPGA). This would allow for a quick assessment of whether this latency is small enough to meet the application's requirements. This would also allow the user to rapidly make major architectural choices and to determine whether the performance bottleneck (and where the improvement efforts should be put) is in the memory hierarchy. Also, since the goal of such a tool would be to speedup development time, it is imperative that the characteristics of communications made by a coprocessor be identified automatically.

With a mix of experimentation and literature, we identified the different variables affecting the performance of data transfers between a coprocessor on an FPGA or SoC-FPGA and its memory. We then developed a methodology to characterize this performance and used it to characterize the communications performance of Xilinx's Zynq-7020 SoC-FPGA.

We then developed a tool to statically determine the memory accesses made by a coprocessor described in HLS C/C++ using the LLVM compiler infrastructure.

Finally, for a series of test cases, the estimated performance (obtained by combining the results of our static memory access analysis and the Zynq-7020 communication performance characterization) was compared to a real implemented system. These tests show that the estimation, while taking only a few dozens of seconds to execute, has an average error of 9% and a maximum error of 25%.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

ACP	Accelerator Coherency Port
ASIP	Application-Specific Instruction-set Processor
API	Application Programming Interface
AST	Abstract Syntax Tree, ou Arbre de Syntaxe absTraite
BB	Bloc de Base
BP	Bande-passante
CFG	Control-Flow Graph, Graphe de flot de contrôle
DMA	Direct Memory Access, unité spécialisée dans la copie de données
FF	Flip Flop (bascule D, élément programmable d'un FPGA)
FPGA	Field-Programmable Gate Array
GEP	*GetElementPtr*, instruction de l'IR LLVM
GPGPU	General-purpose Processing on Graphics Processing Units
HDL	Hardware Description Language (ex. VHDL, Verilog)
HLS	High-Level Synthesis (synthèse de haut-niveau)
IA	Intensité arithmétique
IR	Intermediate Representation (représentation intermédiaire entre un code source et son équivalent en code machine)
LUT	LookUp Table (élément programmable d'un FPGA)
MMU	Memory Management Unit
OCM	On-Chip Memory
PL	Programmable Logic : partie programmable d'un SoC-FPGA
PS	Processing System : processeur fixe d'un SoC-FPGA
RTL	Register Transfer Level
SIMD	Single Instruction Multiple Data (traidement de plusieurs données par une seule instruction
SoC	System-on-a-Chip : système sur puce
FPGA-SoC	SoC incluant une partie FPGA
SSA	Static Single Assignment

STL Standard Template Library, librairie C++ de *templates* standardisée (ISO/CEI 14882).

TLB Translation Lookaside Buffer

TLM Transaction-Level Modeling

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

1.1 Résumé du travail

Nous proposons et évaluons dans ce travail une nouvelle approche d'estimation de la latence des communications entre un coprocesseur sur FPGA et sa mémoire. Cette approche est basée sur une pré-caractérisation automatisée de ce FPGA suivie d'une analyse statique du code source du coprocesseur permettant de déterminer les communications effectuées entre ce coprocesseur et sa mémoire. L'analyse présentée dans ce travail est applicable à des coprocesseurs décrits en C/C++ pour synthèse de haut niveau.

Le but de cette approche est de diminuer significativement le temps nécessaire pour obtenir cette mesure de latence tout en donnant un résultat proche de la réalité. Comme notre approche ne nécessite pas de synthèse ni de simulation du système, elle peut donner une estimation de latence des communications en une ou deux secondes. En mettant en commun cette estimation avec l'estimation du nombre de cycles d'exécution du coprocesseur calculé par l'outil de synthèse de haut niveau utilisé, il est possible d'estimer le temps d'exécution du coprocesseur en l'espace de quelques dizaines de secondes. Ce délai, qui est significativement plus court que même les approches de simulation approximatives existantes, facilite le flot d'amélioration itératif des performances du coprocesseur développé.

De plus, l'approche est agnostique dans le sens qu'elle permet d'obtenir simultanément une estimation de la performance pour tous les modèles de FPGA et pour tous les moyens de communications configurables sur ces mêmes FPGA, pour peu que ceux-ci aient été pré-caractérisés. Ceci permet de guider l'assignation des communications à des ressources matérielles fixes et de valider que la bande-passante disponible sur le FPGA ciblé soit suffisante pour atteindre les requis de performance du design.

Les résultats sont prometteurs : pour un ensemble de 6 cas de tests limités par les communications vers la mémoire et synthétisés à différentes fréquences et avec différentes configurations d'accès à la mémoire, l'erreur relative de l'estimation est en moyenne de 8,5% et varie de 0,2 à 25%. L'exécution de cette estimation est cependant un à deux ordres de grandeur plus rapide qu'une synthèse et implémentation. Des améliorations possibles à la caractérisation des communications et à la méthodologie d'estimation sont aussi suggérées pour améliorer la précision de l'estimation sans augmenter son temps d'exécution.

1.2 Définitions et concepts de base

Cette section comporte l'essentiel nécessaire à la compréhension de cette introduction. Cependant, des explications plus exhaustives seront données dans le chapitre 2.

1.2.1 Accélérateur sur FPGA

Dans le cadre de ce mémoire, nous définissons le terme *accélérateur sur FPGA* comme un coprocesseur à usage spécifique chargé d'accélérer une portion de code d'un logiciel exécuté par un processeur à usage général. Le processeur, lorsqu'il arrive à cette portion de code, communique avec le coprocesseur pour lui indiquer les données à traiter puis est informé lorsque ce traitement est terminé. Cette solution est souvent envisagée en raison des possibilités de parallélisme du FPGA, qui possède en théorie la capacité d'exécuter simultanément toutes portions de code ne comportant pas de dépendances de données.

La communication entre le processeur général et l'accélérateur sur FPGA peut être faite de plusieurs façons (qui sont détaillées à la section 2.2.1 de ce travail). Néanmoins, nous supposerons ici que l'accélérateur et le processeur sont faiblement couplés et qu'ils ont tous deux accès à la même mémoire principale de la puce pour lire et écrire les données. Nous supposerons aussi que le coprocesseur possède également quelques registres de configuration, adressables par le processeur, pour que le logiciel puisse indiquer l'adresse des données à traiter et où écrire le résultat.

1.2.2 Synthèse de haut niveau

La synthèse de haut niveau (HLS) est le processus de transformation de code écrit dans un langage de haut niveau (souvent C/C++) vers une représentation RTL en Verilog ou VHDL. Cette représentation RTL est par la suite passée en entrée à un synthétiseur pour réaliser l'implémentation finale sur FPGA ou ASIC.

De plus, les outils HLS fournissent aussi généralement une estimation de latence d'exécution de la micro-architecture générée. Ceci permet d'obtenir en quelques dizaines de secondes à quelques minutes une estimation de la performance atteignable sans exécution ou simulation, facilitant un flot de développement itératif.

La synthèse de haut niveau et les environnements de développement au niveau du système sont présentés plus exhaustivement à la section 2.5 et 2.5.2 de ce mémoire.

1.2.3 Code C/C++ synthétisable

De manière générale, l'expression *code C/C++ synthétisable* correspond à un code C/C++ qui peut être synthétisé sans erreurs par un logiciel de synthèse de haut niveau, c'est-à-dire qu'il ne :

1. contient pas d'allocation dynamique,
2. fait aucun appel système.
3. contient pas de fonctions récursives.
4. contient pas de pointeurs de fonctions.

Il peut y avoir de légères différences dans les éléments de code pouvant être synthétisés d'un outil de synthèse de haut niveau à l'autre. Dans le cadre spécifique de ce mémoire, l'expression *code C/C++ synthétisable* correspond donc au code C/C++ pouvant être synthétisé par l'outil Vivado® HLS de Xilinx. Des exemples spécifiques à Vivado® HLS de code non-synthétisable sont donnés dans le manuel d'utilisateur du logiciel [1, section *Unsupported C Constructs*, p.300].

1.2.4 FPGA-SoC

Un FPGA-SoC est un SoC comportant une portion FPGA ainsi qu'un processeur et des périphériques généraux. Cette catégorie de FPGA inclut par exemple la série Zynq de Xilinx et le Cyclone V Sx d'Intel FPGA. L'une des particularités de ces FPGA-SoC par rapport aux FPGA traditionnels est la présence de chemins des données fixes entre la portion FPGA et le contrôleur de la mémoire principale ainsi qu'entre le FPGA et le processeur. Ce sujet sera détaillé au chapitre 2.

1.2.5 Métriques de performance

Dans le cadre de ce mémoire, la *performance* du système sera principalement mesurée en débit d'information par unité de temps. On utilisera souvent le terme bande-passante (*memory bandwidth*) dans le sens de débit d'information, comme on le lit régulièrement dans la littérature scientifique et technique. Le contexte et les explications locales détermineront le sens du terme.

Ce travail utilisera aussi le terme *latence*. Il distinguera deux types de latence affectant le temps d'exécution d'un coprocesseur : la latence de la micro-architecture et la latence des communications. La première correspond au temps d'exécution du coprocesseur sans tenir

compte de facteurs externes comme la latence des communications vers la mémoire. Dans le cas de coprocesseurs HLS, ceci correspond à la latence estimée par l'outil de synthèse de haut niveau. La latence des communications correspond au temps nécessaire pour obtenir les données nécessaires à l'exécution du coprocesseur et l'écrire les résultats. Nous définissions aussi la latence d'exécution comme correspondant au temps d'exécution réel du coprocesseur. Celle-ci est fonction de la latence de la micro-architecture et de la latence des communications.

1.3 Importance de considérer et de mesurer la bande-passante vers la mémoire

Les possibilités d'amélioration de performances par coprocesseur sur FPGA sont de prime abord alléchantes. Cependant, un utilisateur cherchant à adapter une portion de code pour l'implémenter en matériel se heurtera rapidement à quelques problèmes majeurs. Notamment, le coprocesseur créé et le processeur étant faiblement couplés, la latence des communications entre les deux peut être significative par rapport au temps d'exécution de la portion d'algorithme synthétisée [2]. Aussi, peu importe le niveau de parallélisme que l'on peut croire possible si l'on ne se fie qu'à la micro-architecture du coprocesseur, ce parallélisme reste limité en pratique par la vitesse à laquelle le coprocesseur peut lire et écrire les données à traiter depuis et vers la mémoire. De plus, alors que les données déjà accédées sont implicitement mises en cache dans le cas d'un logiciel exécuté sur un processeur général, cette cache doit être explicitement décrite dans le code source du coprocesseur. Finalement, les interfaces de communication orientées vers la performance supportent des transactions en rafale, où les communications d'adresses croissantes et contiguës sont significativement plus rapides. Alors que les contrôleurs de cache de processeurs généraux sont conçus pour utiliser ces transactions en rafale le plus possible, le code d'un coprocesseur doit souvent être adapté pour en tirer partie.

Or, les estimations de latence d'exécution fournies par les outils HLS ne modélisent pas ces particularités des communications. En effet, puisque ces outils fonctionnent au niveau de modules (blocs *IP*), il n'ont pas les informations sur le système complet nécessaires pour modéliser plus que l'interface utilisée pour effectuer ces communications. Pour connaître le temps d'exécution réel du module synthétisé, il faut donc simuler ou exécuter tout le système.

Ceci peut mener à deux problèmes. D'abord, cette simulation ou exécution prend beaucoup plus de temps que l'estimation de latence d'exécution fournie par l'outil HLS, qui ne fait qu'une analyse statique du code HDL généré. Dans un monde où un court temps de mise en marché est généralement essentiel, cela décourage un processus itératif d'amélioration des performances qui aurait pu donner une meilleure solution finale. Ensuite, la présence de cette estimation de latence de l'outil HLS peut induire un utilisateur inexpérimenté en erreur en

lui laissant présager une performance significativement supérieure à la réalité. Un corollaire intéressant de ce point est qu'apporter des modifications pour améliorer les communications effectuées peut avoir un impact négligeable sur cette estimation de latence d'exécution, mais significatif dans la réalité.

Ce deuxième problème peut être montré en comparant les deux versions de la convolution 2D 3x3 (filtre de Sobel) présentées à la figure 1.1. La première version (en rouge) est adaptée à une exécution logicielle alors que la deuxième (en vert) est plus adaptée à une synthèse matérielle. La principale différence entre les deux est l'utilisation d'une cache de 4 lignes de l'image, permettant de ne lire en mémoire chaque pixel qu'une seule fois, contrairement à la première version qui lit chaque pixel 9 fois. Si l'on applique une directive de pipelinage à la boucle intérieure de *sobel_filter* et que l'on synthétise les deux versions avec Vivado HLS 2017.4 pour une image de 1920x1080 pixels, cet outil estimera une latence de 16 Mcycles (8 cycles/pixel) pour la première version et 10M cycles (5 cycles/pixel) pour la deuxième. Par contre, si l'on implémente et exécute ces deux versions sur un Zynq-7020, nous constaterons que la première prend en réalité 174 Mcycles, mais que la deuxième ne prend que les 10 M estimés. Nous sommes donc en réalité presque 11x plus lents que l'estimation dans un cas, mais à moins de 1% de celle-ci dans l'autre. Dans le premier cas, le goulot d'étranglement de la performance est la bande-passante vers la mémoire alors que dans le deuxième, c'est la micro-architecture du coprocesseur qui limite la performance, ce que Vivado HLS peut modéliser. La première solution est aussi 17 fois plus lente que la deuxième, même si elle ne fait que 9 fois plus de communications, car celles-ci ne sont pas contiguës et ne peuvent donc pas être effectuées en rafale.

Les communications entre un coprocesseur et un processeur ou la mémoire doivent donc absolument être prises en considération très tôt dans le processus de développement car :

1. L'implémentation de celles-ci peut être sous-optimale et empêcher toute accélération.
2. Même si elles sont implémentées de manière optimale, il peut être impossible d'accélérer l'exécution si le ratio communications/calcul est trop bas.
3. La latence entre le processeur et le coprocesseur peut être plus grande que le temps d'exécution du code sur le processeur.

Bien que dans le premier cas, le problème soit relativement simple à régler, ne nécessitant que des modifications au code du coprocesseur, les deux autres cas sont plus complexes. En effet, la solution du deuxième cas implique de migrer vers une nouvelle plateforme matérielle permettant une plus grande bande-passante. Quant à la solution du dernier cas, elle implique de remettre en question les portions de code ayant été sélectionnées pour une exécution sur

```
-inline uint8_t sobel_operator(const int index, uint8_t * img)
+inline uint8_t sobel_operator(const int row, const int col,
+                              uint8_t img[4][IMG_WIDTH])
 {
     const int8_t x_op[3][3] = { { -1,0,1 }, { -2,0,2 }, { -1,0,1 } };
     const int8_t y_op[3][3] = { { 1,2,1 }, { 0,0,0 }, { -1,-2,-1 } };

     int x_weight = 0, y_weight = 0;
     for (int i = 0; i < 3; i++) {
         for (int j = 0; j < 3; j++) {
-            x_weight += img[index + (IMG_WIDTH*(i-1)) + (j-1)] * x_op[i][j];
-            y_weight += img[index + (IMG_WIDTH*(i-1)) + (j-1)] * y_op[i][j];
+            x_weight += img[(row+i-1)%4][col+j-1] * x_op[i][j];
+            y_weight += img[(row+i-1)%4][col+j-1] * y_op[i][j];
         }
     }
     return (255-(uint8_t)(ABS(x_weight) + ABS(y_weight)));
 }

 void sobel_filter(uint8_t* inter_pix, uint8_t* out)
 {
+    uint8_t lineBuffer[4][IMG_WIDTH];
+
+    for (int i = 0; i < 3; ++i) {
+        for (int j = 0; j < IMG_WIDTH; ++j)
+        lineBuffer[i][j] = inter_pix[i*IMG_WIDTH+j];
+    }
+
     for (int i = 1; i < IMG_HEIGHT-1; ++i) {
-        for (int j = 1; j < IMG_WIDTH -1; ++j) {
-            int fullIndex = i * IMG_WIDTH + j;
-            out[fullIndex] = sobel_operator(fullIndex, inter_pix);
+        for (int j = 0; j < IMG_WIDTH; ++j) {
+            uint8_t out_tmp;
+            int in2Lines = ((i+2) % IMG_HEIGHT) * IMG_WIDTH + j;
+            if (j == 0 || j == IMG_WIDTH -1)
+                out_tmp = 0;
+            else
+                out_tmp = sobel_operator(i, j, lineBuffer);
+            out[i * IMG_WIDTH + j] = out_tmp;
+            lineBuffer[(i+2)%4][j] = inter_pix[in2Lines];
         }
     }
 }
```

Figure 1.1 Différences entre une version appropriée pour un logiciel (en rouge) et appropriée pour un coprocesseur (en vert) d'une convolution 2D.

coprocesseur et de considérer en profondeur le flot des données de l'algorithme à accélérer [2].

Il serait donc important d'avoir un moyen d'analyser rapidement les communications effectuées et d'estimer la latence des communications du coprocesseur développé. Cette analyse devrait être rapide, de manière similaire à l'estimation de latence d'exécution d'outils HLS. En combinant ces deux analyses, il serait alors possible d'estimer le temps d'exécution réel du coprocesseur. Celui-ci correspondrait à la valeur maximale entre la latence des communications et la latence du coprocesseur fournie par l'outil HLS si le coprocesseur lit et écrit les données nécessaires en même temps qu'il exécute son traitement et à la somme de deux s'il lit les données, les traite, puis écrit le résultat.

1.4 Objectifs

Le principal objectif de ce mémoire est de démontrer la faisabilité d'une nouvelle approche d'estimation de la latence des communications entre un coprocesseur sur FPGA et sa mémoire. Cette approche combine une pré-caractérisation de la latence des communications d'un FPGA donné et une analyse statique des communications effectuées par un coprocesseur décrit en C/C++ pour synthèse de haut niveau, ce qui permet d'obtenir une estimation de la latence des communications de ce coprocesseur sur ce FPGA. En la combinant à l'estimation du nombre de cycles d'exécution du coprocesseur fourni par l'outil HLS, il devrait être possible d'estimer la latence d'exécution de ce dernier et ce, sans nécessiter d'exécution ou de simulation du coprocesseur.

Cette approche est schématisée à la figure 1.2. Les éléments en orange ont été développés dans le cadre de ce mémoire. Le terme configuration est utilisé pour représenter un moyen de communication donné sur un FPGA donné.

Un avantage important de cette approche est sa rapidité d'exécution, qui favorise un développement itératif. Un autre avantage est la possibilité d'obtenir d'un seul coup une estimation de la performance atteignable pour tous les modèles de FPGA et chemins des données sur un FPGA-SoC dont la pré-caractérisation des performances de bande-passante a été faite. Ceci permet de faire rapidement des choix architecturaux majeurs comme la sélection du bon modèle de FPGA ou moyen de communication à d'un FPGA-SoC. Finalement, un dernier avantage de notre approche est son côté agnostique, car elle n'est pas liée à un seul modèle ou fabriquant de FPGA.

Pour montrer la faisabilité de cette approche d'estimation de latence des communications, nous ciblerons les objectifs spécifiques suivants :

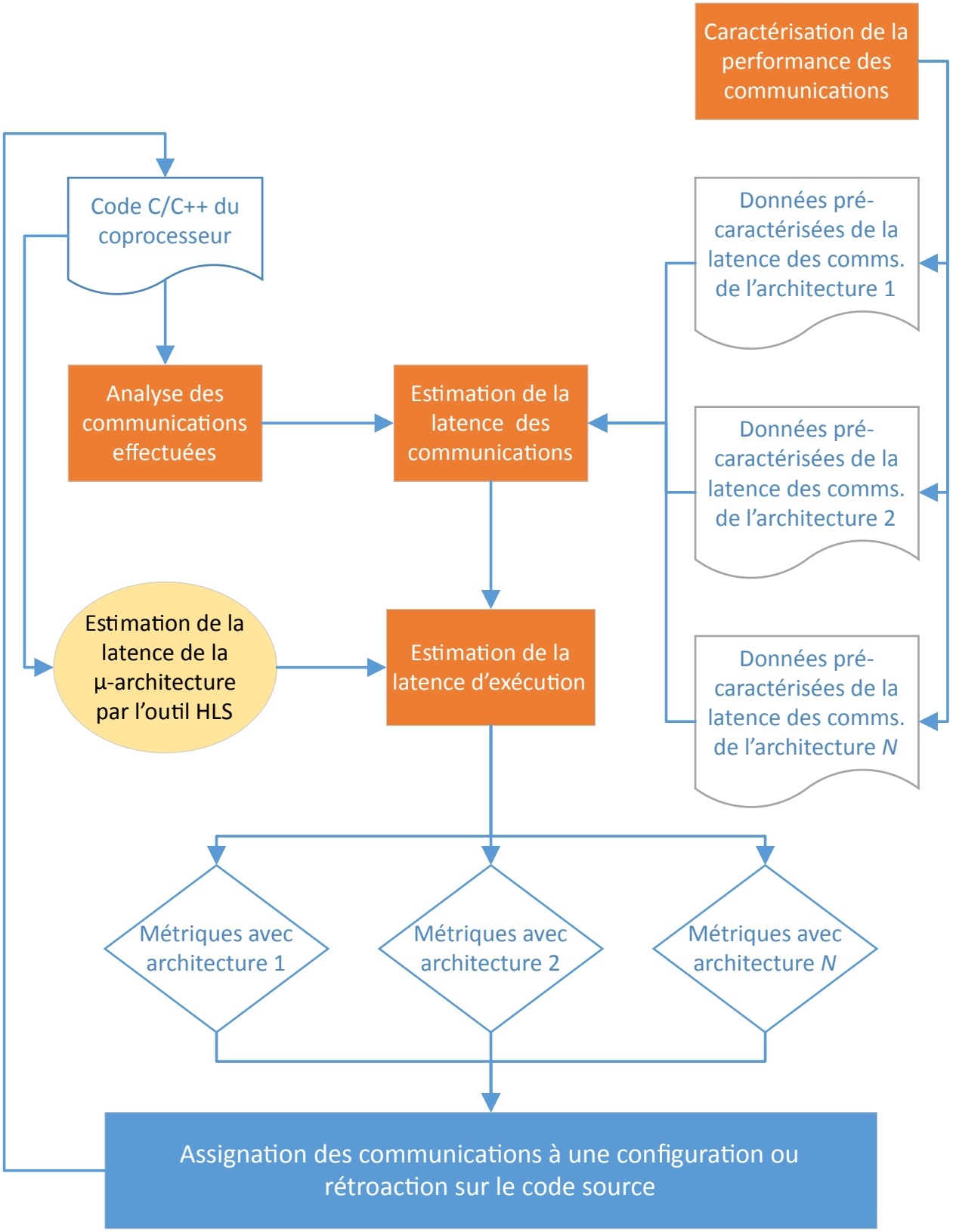


Figure 1.2 Flot proposé d'un estimateur statique de la latence d'exécution d'un coprocesseur sur FPGA.

1. Déterminer les facteurs qui influent la latence.
2. Établir une méthodologie permettant d'obtenir des données expérimentales de latence des communications vers la mémoire d'un coprocesseur donné sur un FPGA donné en fonction de ces facteurs.
3. Tester cette méthodologie sur un FPGA-SoC (Zynq-7020 de Xilinx).
4. Déterminer quelles métriques dépendent de caractéristiques du coprocesseur.
5. Automatiser l'extraction de ces caractéristiques à partir du code source C/C++ du coprocesseur.
6. Combiner les données expérimentales obtenues aux caractéristiques analysées.
7. Comparer les résultats, pour plusieurs cas de test, de cette combinaison aux résultats obtenus par la synthèse et exécution de ces cas de test.

1.5 Contributions

Les contributions de ce travail sont la proposition d'une :

1. Méthodologie pour acquérir de manière automatisée des données expérimentales de performance des communications d'un FPGA ou FPGA-SoC vers une mémoire vive externe.
2. Nouvelle approche par analyse statique permettant d'obtenir d'identifier les communications faites par un coprocesseur décrit en C/C++ synthétisable.
3. Approche améliorée d'analyse statique qui combine les points 1 et 2 pour permettre l'estimation rapide de la performance d'un accélérateur sur FPGA.

Le premier point n'est pas nouveau : la littérature regorge d'articles détaillant la performance espérée des communications de plusieurs modèles de FPGA et FPGA-SoC ainsi que les facteurs affectant cette performance. Cependant, la méthodologie que nous présentons ici est à notre connaissance plus détaillée et plus automatisée que ce qui est disponible publiquement et permet donc de répéter ces expérimentations avec d'autres modèles de FPGA ou FPGA-SoC. De plus, bien que les données présentées dans la littérature permettent d'avoir une idée générale de la situation, elles n'étaient pas assez complètes pour nos besoins. Le deuxième point est à notre connaissance nouveau. Finalement, le dernier point est similaire à l'approche employée par l'environnement de développement SDSoC™ de Xilinx, mais est plus précis. En effet, SDSoC estime les communications uniquement à l'aide de la taille les données accédées par le coprocesseur, ce qui donne un résultat beaucoup moins exact dans plusieurs cas.

1.6 Plan du mémoire

Ce mémoire est divisé en 6 chapitres. Le chapitre 2 de ce mémoire présente une revue détaillée des concepts nécessaires à la réalisation du travail. Celle-ci traite principalement de l'importance de prendre en compte la bande-passante vers la mémoire lorsque l'on tente d'accélérer un algorithme, de la performance de la bande-passante attendue de différents chemins des données vers la mémoire du Zynq-7000, de la synthèse de haut niveau et de l'analyse statique de code C/C++.

La relation entre chapitres 3 à 5 est résumée à la figure 1.3. Le chapitre 3 commence par présenter la méthodologie suggérée pour mesurer la latence des communications entre des coprocesseurs sur FPGA et FPGA-SoC et la mémoire. Nous verrons ensuite comment cette méthodologie a été appliquée pour mesurer la latence des communications d'un coprocesseur sur un Zynq-7020 de Xilinx. Quant au chapitre 4, il présente *HLSComms*, un outil d'analyse statique prenant en entrée un code C/C++ modélisant un coprocesseur et ressortant les caractéristiques des communications effectuées par ce dernier.

Puis, le chapitre 5 présente les résultats obtenus en combinant les travaux présentés aux chapitres 3 et 4 pour estimer la latence d'exécution de plusieurs coprocesseurs de tests. Finalement, le chapitre 6 apporte une conclusion et présente les améliorations à apporter pour les travaux futurs.

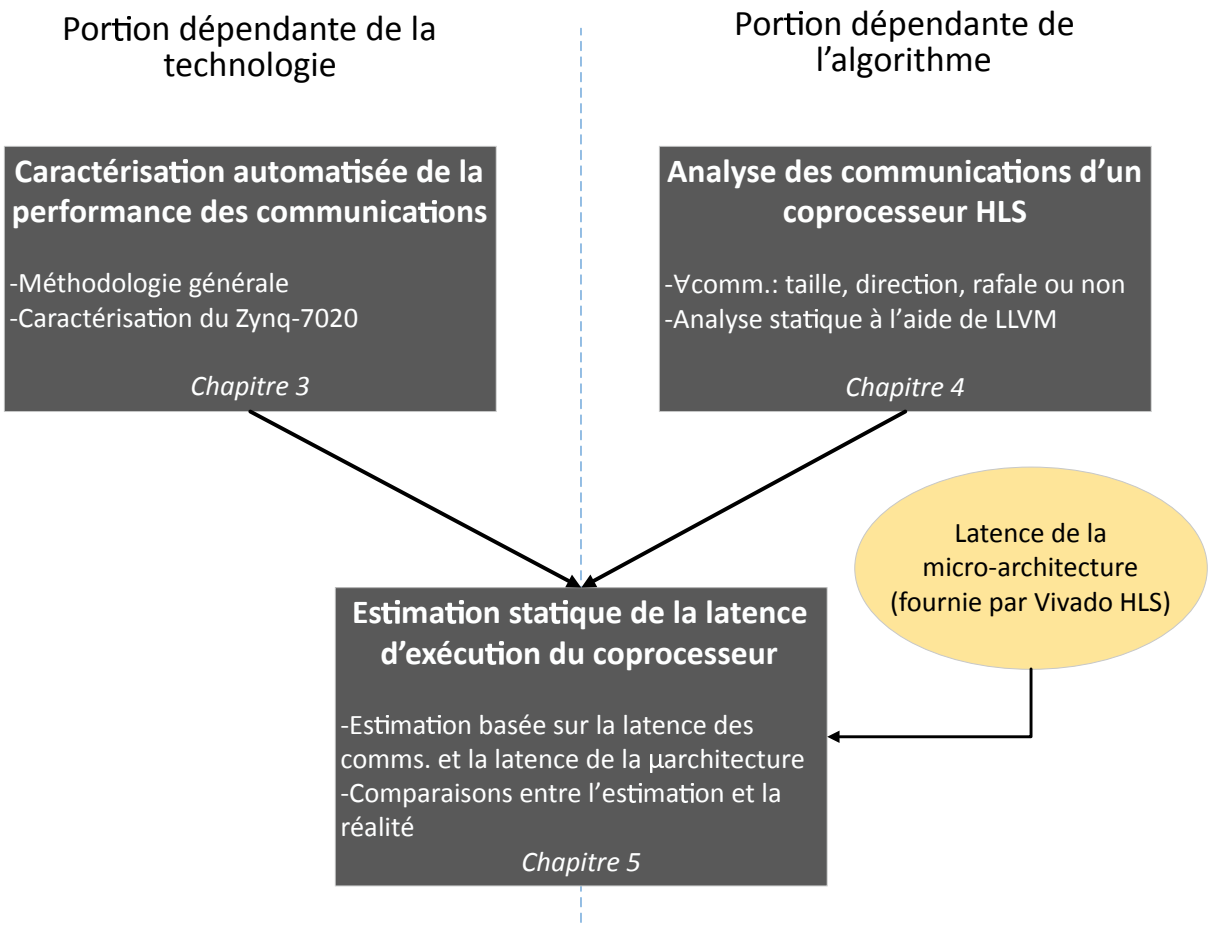


Figure 1.3 Sujets abordés aux chapitres 3, 4 et 5 et relations entre ces derniers.

CHAPITRE 2 REVUE DES CONCEPTS

Cette section présente une revue des concepts nécessaires à la compréhension des chapitres suivants du mémoire afin de mieux situer la contribution du travail par rapport à l'état de l'art.

Nous commencerons par une brève présentation du domaine d'application de l'accélération matérielle d'algorithmes. Puis nous nous pencherons sur la problématique du transfert des données d'un processeur général au coprocesseur chargé de cette accélération, dont la modélisation est le sujet principal de ce travail. Nous présenterons les différents chemins des données utilisés dans des FPGA-SoC, ainsi que quelques solutions à certains problèmes liés à ce transfert de données. Nous terminerons par un aperçu de la synthèse de haut-niveau, des outils existants d'analyse de code source et des outils de transformation source-à-source pour synthèse de haut-niveau existants.

2.1 Domaine d'application de l'accélération matérielle par coprocesseurs personnalisés

La réalisation d'un accélérateur personnalisé sur FPGA (ou ASIC) servant à accomplir une tâche spécifique est beaucoup plus complexe et moins flexible que la simple programmation d'un processeur à usage général ou d'un GPU. En contrepartie, le FPGA permet de réduire la consommation de courant nécessaire à l'exécution de la tâche spécifiée. Cet avantage des FPGA en a fait une solution privilégiée dans le domaine des systèmes embarqués, notamment pour la vision par ordinateur [3], le traitement vidéo [4] ou, plus récemment, l'inférence de réseaux de neurones [5].

De plus, avec la fin de la loi de Moore et surtout de la mise à l'échelle de Dennard, les coûts d'électricité des centres de données augmentent plus rapidement que la performance atteignable des processeurs généraux, ce qui rend l'accélération matérielle attrayante depuis quelques années pour des applications fonctionnant sur des milliers de serveurs. Cette avenue a donc été énormément développée dans les cinq à sept dernières années. Notons par exemple qu'une partie significative du classement des résultats retournés par *Bing* est exécutée sur FPGA [6], de même que des portions de bases de données NoSQL [7], etc. Google, réalisant qu'il faudrait doubler la capacité de calcul de ses centres de données si leurs utilisateurs se servaient de reconnaissance vocale pour aussi peu que trois minutes par jour, a pris la décision de développer un accélérateur d'inférence de réseaux neuronal sur ASIC [8]. Finalement, la

majorité des nouveaux serveurs des centres de données de Microsoft installés dans les deux dernières années comportent des FPGA connectés à la fois à un processeur général hôte par PCI-E et au reste du centre de données par Ethernet [9].

2.2 Importance de la performance du transfert de données

Si l'on veut obtenir une accélération d'un code logiciel exécuté sur CPU en l'implémentant comme coprocesseur, le temps de copier les données vers ce dernier et de réécrire le résultat de manière accessible par le processeur doit être inférieur au temps que le processeur aurait pris pour faire lui-même le calcul. Or, ce temps de transfert n'est pas souvent négligeable. Après une brève introduction des manières possibles d'effectuer ce dit transfert, nous verrons à quel point il est important de le considérer.

2.2.1 Transferts de données entre l'accélérateur sur FPGA et la mémoire

L'utilisation d'un coprocesseur pour accélérer un algorithme implique un transfert de données entre le processeur et le coprocesseur. De manière générale, si processeur et coprocesseur sont sur la même puce, quatre moyens principaux existent pour effectuer cette tâche. Ces façons de faire, représentées aux figures 2.1a à 2.1d, sont les suivantes :

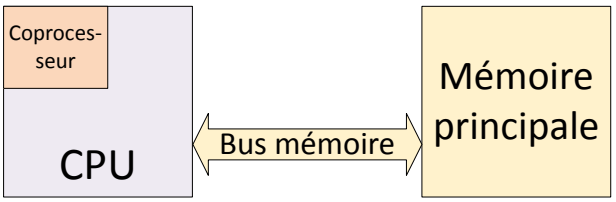
(a) Le coprocesseur et le CPU sont fortement couplés, le premier ayant directement accès au chemin des données du deuxième. Ceci correspond par exemple à des instructions personnalisées sur un ASIP.

(b) Les registres du coprocesseur sont adressables via le bus mémoire du processeur. Ce dernier utilise les mêmes instructions de lecture/écriture que pour charger ou écrire des données en mémoire principale.

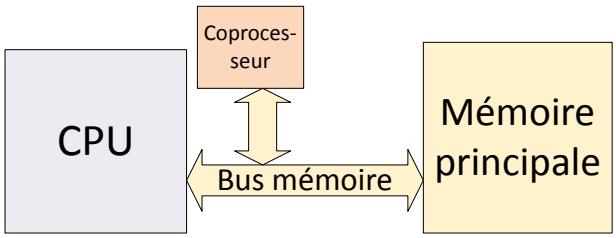
(c) Le CPU utilise une unité de transfert dédiée, le DMA, pour copier des plages de données. L'arrangement du coprocesseur est identique au point précédent.

(d) Le coprocesseur accède lui-même directement à la mémoire principale (possiblement à partir d'un bus dédié), selon ses besoins. Il peut possiblement conserver quelques registres adressables comme interface de configuration.

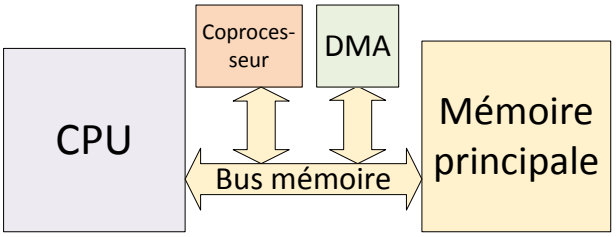
La première approche offre la plus faible latence d'accès aux données et est donc la plus appropriée pour effectuer un grand nombre de petites opérations indépendantes nécessitant chacune une petite quantité de données. Cette approche implique par contre un couplage fort entre le processeur et le coprocesseur : si ce dernier est trop complexe, il deviendra le chemin critique du processeur, diminuant la fréquence atteignable et donc la performance de



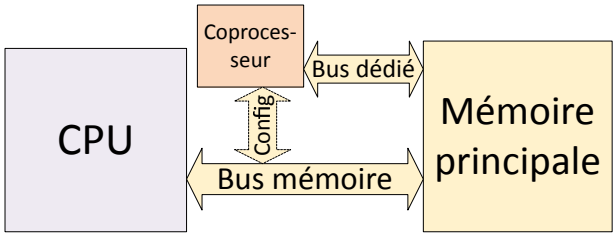
(a) Coprocesseur dans le chemin des données du processeur.



(b) Coprocesseur adressable, transferts par le processeur.



(c) Coprocesseur adressable, transferts par DMA.



(d) Coprocesseur indépendant. Registres de configuration adressables optionnels.

Figure 2.1 Architectures communes de transfert de données entre CPU et accélérateur sur FPGA.

l'ensemble. La deuxième approche, quant-à-elle, est plus appropriée pour la configuration de coprocesseurs, mais est médiocre pour le transfert d'une grande quantité de données, n'offrant généralement qu'une vitesse de transfert faible et monopolisant le processeur pendant l'exécution du coprocesseur. La troisième approche permet généralement un meilleur débit de données et décharge le processeur principal, au détriment d'une latence supplémentaire d'initialisation du transfert. Finalement, la dernière approche permet de combiner la performance de l'approche précédente tout en offrant une plus grande flexibilité au coprocesseur, qui peut alors déterminer lui-même les accès en lecture/écriture qu'il doit faire. Outre cette flexibilité, cela offre aussi la possibilité au coprocesseur d'avoir un chemin d'accès dédié vers le contrôleur mémoire, ce qui peut être plus rapide. C'est le cas par exemple avec le Zynq-7000 vu à la section 2.4.

Le présent projet s'intéresse principalement à la quatrième méthode et considère, à moins d'avis contraire, que le coprocesseur a un chemin dédié vers la mémoire principale et quelques registres de configuration adressables par le processeur général.

2.2.2 Mur de la mémoire

La performance de la bande-passante est considérée comme l'un des principaux goulots d'étranglement possibles de la performance des ordinateurs depuis deux décennies. En effet, la performance de calcul de ceux-ci augmentant significativement plus rapidement que la performance de leur mémoire, un éventuel "mur de la mémoire" avait été prévu il y a plus de 20 ans [10]. Celui-ci aurait imposé une limite pratique aux performances atteignables d'un processeur, peu importe la capacité de calcul de celui-ci. L'argumentaire de l'époque relevait entre autres que ce mur arriverait même avec une cache "idéale", de taille infinie et accessible en un cycle d'horloge, puisque les données doivent tout de même être chargées au moins une fois à partir d'une mémoire externe avant d'être accessibles en cache. Cette prévision s'est majoritairement réalisée, d'après un retour sur celle-ci par l'un de ses auteurs en 2004. Ce dernier notait que bien que l'on ne voyait pas encore ce mur dans tous les domaines, certains domaines d'applications comme des applications transactionnelles ou de calcul scientifique voyaient le processeur inactif 65 et 95% du temps (respectivement), majoritairement en raison d'attente après des données en mémoire [11].

Bien sûr, la progression de la performance scalaire des processeurs a également ralenti significativement au cours des 20 dernières années, ce qui n'avait pas été prévu par les auteurs de la proposition précédente. Cependant, cette baisse de croissance fut palliée par la généralisation d'instructions vectorielles, des processeurs multi-cœurs et du calcul sur GPU et sur FPGA (dans une moindre mesure jusqu'à présent), qui augmentent ainsi significativement la quantité de données nécessaires à chaque cycle d'exécution. En conséquence, les manuels d'architecture d'ordinateurs enseignent qu'il est généralement impossible d'obtenir une bonne performance vectorielle sans fournir un débit de données suffisant et qu'il est important de tenir compte de la latence d'initialisation du processeur vectoriel, un petit nombre de calculs pouvant être plus rapide sur un processeur scalaire général [12].

2.2.3 Modèle en pente de toit

L'impact de la limitation de performance amenée par ce mur de la mémoire dépend de la quantité d'accès à la mémoire effectués par l'algorithme étudié : la performance d'un algorithme faisant plusieurs accès mémoire par opération de calcul sera forcément plus limitée par le débit des données que celle d'un algorithme faisant plusieurs calculs successifs sur les mêmes données. Cette constatation peut être formalisée par le modèle en pente de toit [13]. Ce modèle visuel, tiré du monde du calcul haute-performance et montré graphiquement à la figure 2.2, met en relation la bande-passante (BP) vers la mémoire et la performance de calcul (PC) du processeur utilisé, le tout fonction de l'*intensité arithmétique* (IA), métrique

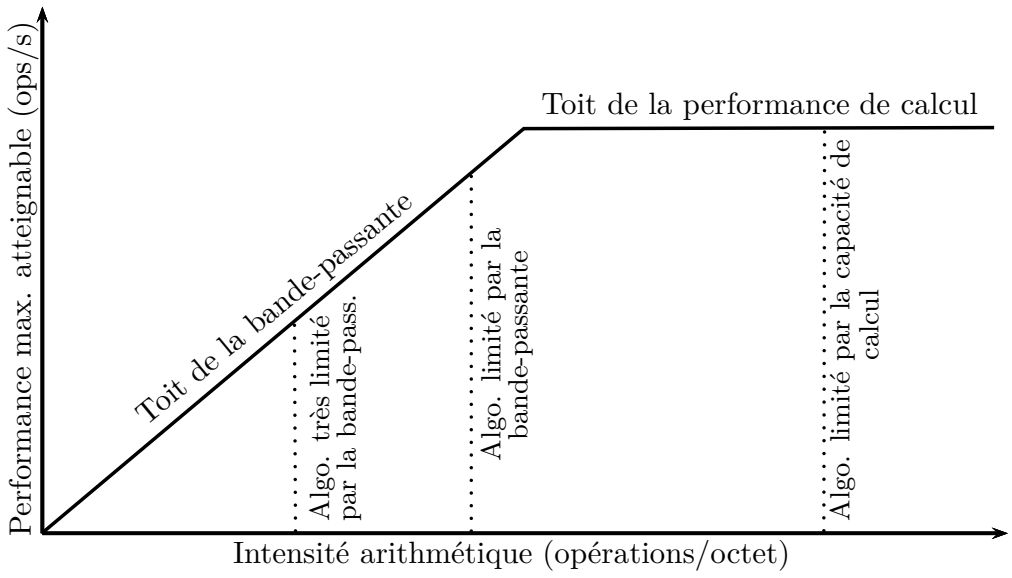


Figure 2.2 Modèle en pente de toit

propre à l'algorithme étudié qui correspond au ratio du nombre d'opérations effectuées sur la quantité de données accédées. Ainsi, le modèle, résumé à l'équation 2.1, indique que pour une faible intensité arithmétique (relativement beaucoup d'accès mémoire par opération de calcul), le facteur limitant la performance est la bande-passante et qu'il est alors inutile de tenter d'augmenter cette performance en augmentant la performance de calcul (par exemple à l'aide d'instructions vectorielles, etc.). Par contre, une fois une certaine intensité arithmétique atteinte, la limitation devient la capacité de calcul du processeur utilisé. Notons que la performance maximale de calcul et la bande-passante disponible dépendent de la plateforme utilisée.

$$Perf.\ max._{\mathrm{CPU}} = \min \begin{cases} PC \\ IA \times BP \end{cases} \tag{2.1}$$

Une extension de ce modèle pour tenir compte de la flexibilité architecturale des FPGA a été proposée par da Silva et al. [14], qui suggèrent de séparer les ressources du FPGA en éléments de calculs (EC) contenant tout le nécessaire pour exécuter l'algorithme étudié et à la performance de calcul $\mathrm{PC_{EC}}$. Puis, sachant que cet élément de calcul peut être dupliqué autant de fois que les ressources du FPGA le permettent, ils suggèrent un facteur d'extensibilité (EX), défini par :

$$SC = \frac{Ressources\ disponibles}{Ressources\ par\ EC} \tag{2.2}$$

La performance de calcul maximale devient alors

$$PC_{\text{FPGA}} = PC_{\text{EC}} \times EX \tag{2.3}$$

et la performance atteignable

$$Perf.\ max._{\text{FPGA}} = \min \begin{cases} PC_{\text{EC}} \times EX \\ IA \times BP \end{cases} \tag{2.4}$$

En comparant les équations 2.1 et 2.4, il ressort que bien que le FPGA offre la possibilité d'augmenter le nombre de ressources utilisées pour augmenter sa performance de calcul, la limitation de bande-passante est la même. Ainsi, en supposant une bande-passante comparable entre un processeur et un FPGA (ce qui est notamment le cas de plusieurs FPGA-SoC), le seul moyen d'accélérer un algorithme à faible intensité arithmétique est de modifier la plateforme utilisée pour obtenir une meilleure bande-passante. Cette dernière option doit cependant être considérée très tôt dans le développement, puisqu'elle implique des changements matériels importants. Il est donc primordial de connaître ses requis en bande-passante avant de fixer la plateforme utilisée. Malheureusement, nous avons vu ce point négligé à quelques reprises : même des ingénieurs de Google admettent avoir sous-estimé l'impact de la bande-passante externe lors de la réalisation de coprocesseurs pour l'inférence de réseaux de neurones. Ils notent que 90% de ses applications sont limitées par cette bande-passante, bien que des technologies plus rapides existaient au moment de la conception [8].

Un corollaire intéressant des limitations possibles causées par la bande-passante est amené par Naylor et al. [15], qui suggèrent de considérer l'utilisation d'un simple processeur vectoriel plutôt que de développer un accélérateur spécifique si l'algorithme à accélérer a une intensité arithmétique faible. Cette suggestion provenait de la comparaison de performance, pour une même application de calcul neuronal, d'un coprocesseur fait sur mesure et d'une simple extension d'instructions vectorielles personnalisées au jeu d'instructions d'un processeur général. Alors que le premier avait nécessité trois années de travail et n'était approprié que pour cette application particulière, le deuxième a été réalisé beaucoup plus rapidement et a notamment pu être adapté à un deuxième modèle neuronal en l'espace de deux jours. Pourtant, l'algorithme étant limité par la mémoire, les deux accélérateurs offraient une performance comparable en utilisant une quantité de ressources comparables.

2.2.4 Latence d'initialisation

Outre la bande-passante entre le coprocesseur et ses données, le processeur doit configurer le coprocesseur pour contrôler son exécution et ce dernier doit aviser le processeur une fois l'exécution terminée. Cette latence n'est pas négligeable si la quantité de données traitées est faible et peut empêcher toute accélération. Notamment, des travaux précédents de notre laboratoire constatent, pour une application de traitement vidéo sur une plateforme donnée, que le temps d'initialisation et de transfert d'une portion d'image de 16x16 pixels est comparable au temps d'initialisation et de transfert d'une image complète de 256x256 pixels [2]. De même, Xilinx indique que, pour le Zynq-7000, cette latence peut avoir un impact significatif sur des transferts de moins de 4 ko [16, p. 656].

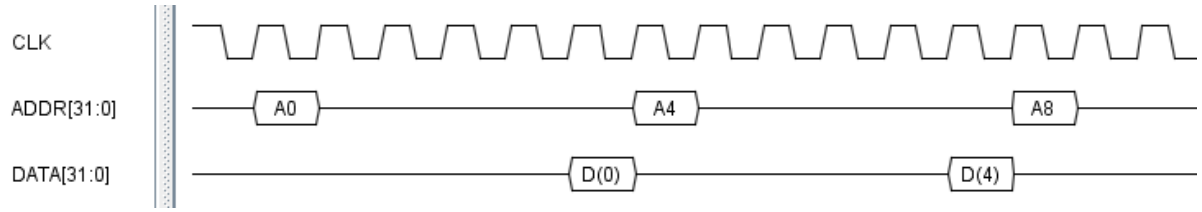
2.2.5 Transferts en rafale

La quasi-totalité des protocoles de communications modernes orientés vers la performance de la bande-passante offrent l'option d'effectuer des transferts en rafale, qui permettent de transférer plusieurs données adressées successivement en ne transmettant l'adresse que pour la première donnée. Ceci est particulièrement avantageux dans des situations où la latence entre la requête d'une donnée et sa réception est significative (par exemple, avec une mémoire externe), puisque cette latence est alors masquée par la réception de plusieurs données successives. Une comparaison simplifiée d'un transfert avec et sans rafales est présenté à la figure 2.3. On note la performance significativement meilleure du transfert avec rafale, qui n'a pas besoin d'attendre de recevoir la requête d'une deuxième donnée avant de la transférer.

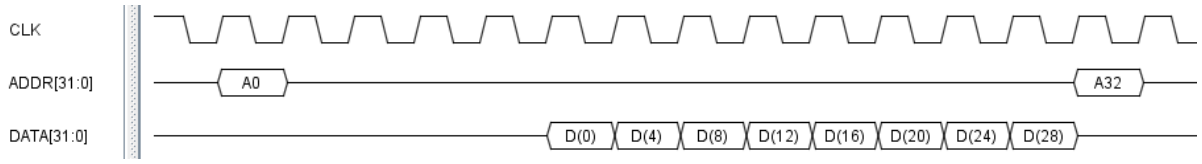
La différence de performance atteignable est significative. Par exemple, nos tests (détaillés aux chapitre 3) montrent, pour un transfert de 128 ko sur un Zynq-7020, une bande-passante atteignable différente d'un facteur 10 selon que les données soient envoyées en rafale ou non. Un constat encore plus flagrant a été fait sur un Virtex 7 connecté par PCI-E à un processeur x86, où la différence atteint un facteur de 88 avec 70 Mo/s sans rafales et 6,2 Go/s avec rafales [17].

2.2.6 Flot des données sur coprocesseur

Outre les possibles limitations de performance mentionnées plus haut, la programmation de coprocesseurs sur FPGA (ou ASIC) impose des difficultés supplémentaires due à l'absence de hiérarchie de mémoire implicite à l'utilisation. En effet, lorsque l'on programme un logiciel pour processeur général, on peut s'attendre à ce que les données accédées fréquemment se retrouvent en cache, contrairement à une implémentation matérielle où une telle cache



(a) Transferts sans rafales



(b) Transferts avec rafales

Figure 2.3 Transferts (très simplifiés) sans et avec rafales

doit être explicitement décrite. Ceci permet de développer une cache adaptée spécifiquement aux besoins spécifiques de l'algorithme accéléré mais cela implique aussi qu'une attention particulière doit être portée au flot des données, sans quoi les performances atteignables seront significativement limitées.

Malheureusement, sous-estimer ou ignorer cette nécessité pour se concentrer sur l'architecture de l'accélérateur reste une erreur fréquente dans la littérature [18]. Cette erreur est peut-être partiellement due au fait que plusieurs outils se concentrent uniquement sur le développement du coprocesseur, s'arrêtant aux interfaces d'entrées/sorties. Ils ne peuvent donc pas modéliser correctement la latence des accès en mémoire, puisque celle-ci dépend de comment ce coprocesseur est intégré dans le reste du système.

Cette attention devant être portée au flot des données est également l'un des obstacles majeurs à l'adaptation de portions de code logiciel pré-existant vers un coprocesseur matériel, empêchant toute accélération dans certains cas [19]. Une méthodologie cherchant à faciliter une telle conversion devrait donc en priorité porter attention à ce flot des données [2].

2.2.7 En résumé

En résumé, les 4 éléments suivant doivent être pris en compte pour réaliser un accélérateur efficace :

1. L'intensité arithmétique de l'algorithme doit être élevée.
2. Il est nécessaire d'établir une hiérarchie de mémoire efficace.
3. Les accès à la mémoire principale doivent être effectués en rafale.

4. La latence d'initialisation du coprocesseur doit être négligeable par rapport à son temps exécution.

2.3 Contraintes additionnelles pour le transfert de données

Aux contraintes amenées par la bande-passante et la latence d'accès aux données s'ajoutent certaines contraintes pratiques qui complexifient également ce transfert : notamment, il faut s'assurer de la cohérence entre la cache du processeur et les données utilisées par le coprocesseur et, si le processus contrôlant le coprocesseur utilise des adresses virtuelles, celles-ci doivent être traduites correctement. Ces contraintes, qui sont rencontrées par tout système hétérogène, ont des solutions qui seront détaillées ici.

2.3.1 Cohérence de cache

L'un des problèmes du partage de données entre processeur général et coprocesseur survient lorsqu'il y a une différence entre les données dans la cache du processeur et celles de la mémoire principale, soit parce qu'un coprocesseur a modifié cette dernière ou parce que le processeur écrit dans une cache en mode d'écriture différée. Les solutions courantes pour éviter ce problème sont :

1. De marquer la plage d'adresses partagée comme non-cacheable, forçant chaque accès du processeur à se rendre à la mémoire principale.
2. De vider la cache pour la plage mémoire partagée avant de lancer le traitement sur le coprocesseur.
3. D'avoir un mécanisme matériel assurant la cohérence de cache.

Chacune de ces solutions a ses avantages et inconvénients : le choix optimal dépend donc du contexte d'utilisation. Par exemple, la première convient bien à des données qui ne sont pas destinées au processeur mais plutôt à un autre périphérique, comme un tampon de trames pour un affichage vidéo. La deuxième solution augmente la latence d'exécution du coprocesseur, ce qui peut être particulièrement problématique dans le cas où ce dernier a un court temps d'exécution sur peu de données. Finalement, avoir un mécanisme matériel de cohérence de cache permet de simplifier la programmation du processeur et d'éviter la pénalité de latence qu'apportent les deux premières solutions. Cela permet aussi, si les données accédées par le coprocesseur sont déjà en cache, de réduire la latence d'accès à ces données depuis ce dernier et de diminuer la consommation d'énergie du transfert [20].

Cependant, cette dernière méthode peut aussi réduire les performances du processeur si la quantité de données accédées est dans le même ordre de grandeur (ou plus) que la taille de la cache en forçant l'éviction de données utiles à celui-ci. De plus, cela limite la bande-passante vers la mémoire à celle du contrôleur de cache, qui peut être plus limitée que celle d'un contrôleur indépendant car le premier est orienté vers les besoins relativement limités en débit (au bénéfice de la latence d'accès) des processeurs à usage général. Ceci est notamment le cas pour certains FPGA-SoC comme le Zynq-7000 et le Cyclone V SE [21], tel que détaillé à la section 2.4.4.

Finalement, notons que les mécanismes matériels assurant la cohérence de cache ne sont pas nouveaux : ils sont une nécessité pour les systèmes multiprocesseurs ayant au moins un niveau de cache privé. Nous verrons comment elle est implémentée dans la majorité des systèmes basés sur des processeurs ARM puis, côté centre de données, pour certains accélérateurs dédiés.

ARM Accelerator Coherency Port

Plusieurs systèmes multiprocesseurs ARM, comme le Cortex-A9 MPCore présent dans les FPGA-Soc Zynq-7000 de Xilinx et Cyclone-V Sx d'Intel FPGA, connectent les différents processeurs au système mémoire via le *Snoop Control Unit* (SCU) [22, p. 25]. Celui-ci s'interpose entre la cache L1 privée des processeurs et une cache L2 partagée du système et s'occupe, entre autres, d'assurer la cohérence de cette dernière cache entre les différents processeurs. Il est possible d'ajouter un port *AXI* 64 bit esclave supplémentaire au SCU, l'*Accelerator Coherency Port* (ACP), qui permet de connecter un co-processeur au système géré par le SCU. Ce co-processeur peut alors accédéer directement à la mémoire de la même manière que les processeurs le feraient, garantissant la cohérence de cache au passage. Les concepteurs de FPGA-SoC de Xilinx et Intel ont connecté l'ACP à la partie programmable du SoC [16], [23], permettant des accès cohérents par les accélérateurs sur FPGA. Un résumé des interconnexions du SCU du Zynq-7000 est présenté à la figure 2.4. L'architecture de cette partie du chemin des données est similaire pour le Cyclone V Sx.

Cohérence avec accélérateurs externes

Des solutions de cohérence de cache existent aussi dans le cas d'accélérateurs situés sur une autre puce que le processeur qui le contrôle. Encore une fois, il s'agit principalement d'adaptations de solutions pré-existantes du domaine des processeurs généraux ou de l'accélération par GPGPU.

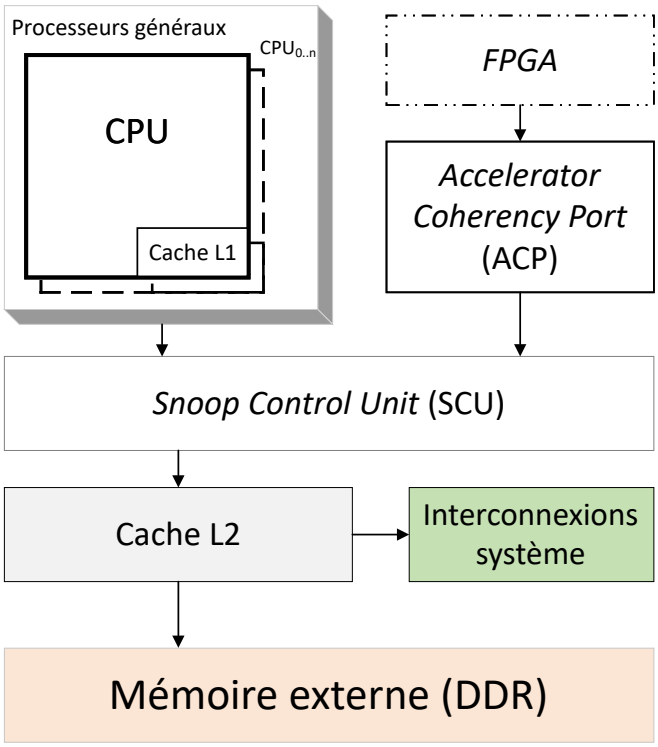


Figure 2.4 Modèle simplifié du *Snoop Control Unit* et de l'ACP sur le Zynq-7000.

Par exemple, la plateforme QAP d'Intel [24], permet de connecter un FPGA à la place d'un processeur x86 d'Intel sur une carte-mère supportant plusieurs processeurs. Le FPGA et le CPU communiquent alors de la même façon que deux CPU le feraient. Une approche différente, bâtie sur mesure pour des accélérateurs personnalisés, est présente dans les processeurs POWER8 et POWER9 d'IBM, qui comportent un *Coherent Accelerator Processor Interface* (CAPI) [25] qui permet un accès cohérent à la mémoire du processeur par un système sur carte PCI-E.

Une évaluation quantitative d'accélérateurs avec et sans transferts cohérents depuis le processeur a aussi été réalisée par Choi et al. [26]. Celle-ci montre que comme avec des accélérateurs sur la même puce que le processeur, outre une simplicité de programmation, les accès cohérents réduisent la latence d'initialisation du transfert, ce qui avantage les transferts relativement petits. Cependant, la bande-passante atteignable est généralement inférieure à celle d'une solution où l'accélérateur possède sa propre mémoire dédiée, ce qui peut rendre cette deuxième solution plus avantageuse si la quantité de bande-passante requise est élevée.

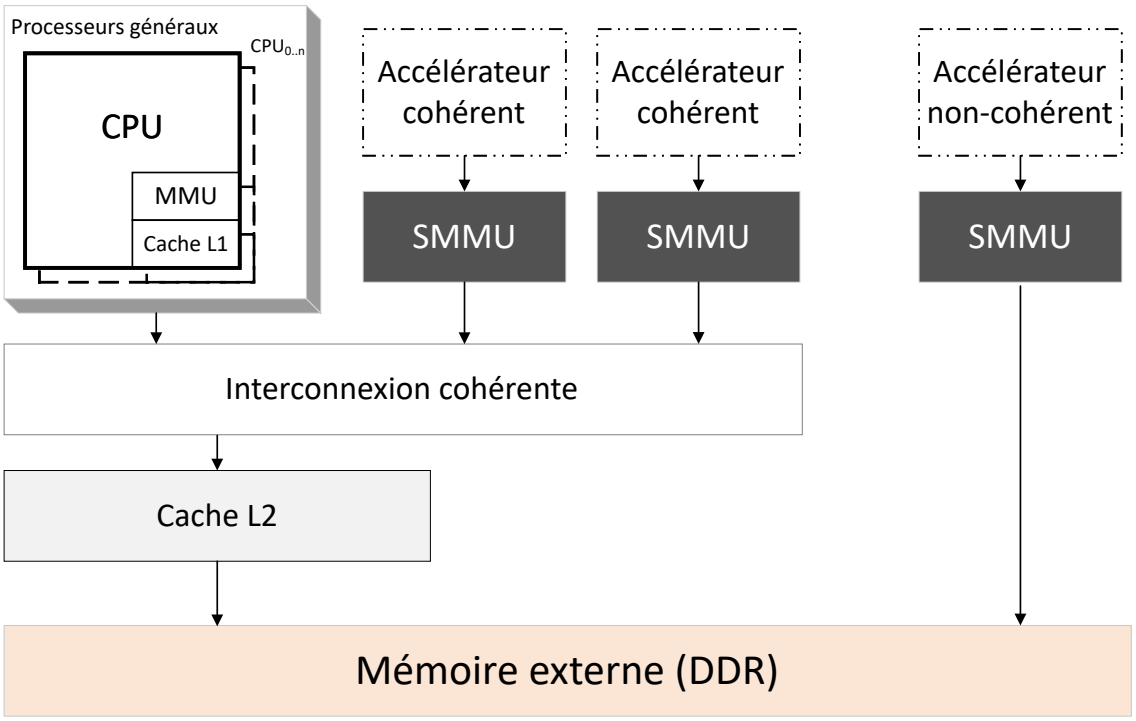


Figure 2.5 Représentation simplifiée d'un système embarqué avec comportant un SMMU.

2.3.2 Adresses virtuelles

Un autre problème pratique du transfert de données entre processeur et FPGA est l'utilisation dans la majorité des cas d'adresses virtuelles par le premier. Outre le fait qu'une traduction vers une adresse physique doit être faite pour qu'un accélérateur connecté à la mémoire puisse accéder aux bonnes données, les adresses virtuelles ne sont que contiguës par pages : deux pages virtuellement successives peuvent être discontinues en mémoire.

L'une des principales solutions "classiques" à ce problème est de s'assurer, par un pilote au niveau du système d'exploitation, que l'allocation est faite sur des pages physiquement contiguës. Cette solution nécessite toutefois l'utilisation d'un API distinct pour allouer la mémoire, ce qui peut compliquer la programmation et peut être problématique dans quelques cas, notamment lorsque la mémoire utilisée est allouée par un processus externe. L'accélérateur doit dans ce dernier cas être modifié pour supporter les transferts par pages, au détriment de la simplicité et de la performance.

Comme avec la cohérence de cache, une solution matérielle existe pour faciliter les transferts : il suffit d'ajouter un MMU au niveau du système, plutôt que seulement au niveau du processeur. Ceci permet entre autres de partager les même tables de traduction entre le processeur et le coprocesseur, permettant au deuxième d'utiliser directement les adresses virtuelles du

premier. ARM appelle son implémentation de cette fonctionnalité le *System Memory Management Unit* (SMMU) [27] et celle-ci est incluse dans quelques FPGA-SoC récents comme le Zynq Ultrascale [28, p.77] de Xilinx et le Stratix-10 [29] d'Intel FPGA. Une représentation simplifiée d'un SMMU dans un système embarqué est présenté à la figure 2.5.

Cette solution matérielle vient cependant avec un coût en performance : la traduction d'adresse n'est pas instantanée, surtout si cette donnée n'est pas dans le TLB. De plus, d'après Hao et al. [30] les MMU existant présentement ont été conçus pour maximiser la performance de patrons d'accès mémoires typiques d'un processeur général ou d'un GPU, qui sont différents de ceux habituellement rencontrés dans un accélérateur, ce qui résulte en une performance limitée pour la grande majorité des accès. Des solutions ont été proposées par ces auteurs ainsi que par Vogel et al. [31], mais, à notre connaissance, au moment d'écrire ces lignes, elles n'ont pas encore été implémentés dans aucune solution commerciale.

2.4 Architecture de communications du SoC-FPGA Zynq-7000 de Xilinx

Cette section détaille les différents chemins des données pouvant être utilisés pour la communication entre la partie programmable (PL) et la partie processeur (PS) du Zynq-7000. Dans un premier lieu, nous décrirons qualitativement chacun de ces chemins, puis nous détaillerons la performance espérée de ceux-ci en se basant sur une combinaison de la littérature et de tests réalisés pour le présent travail.

2.4.1 Descriptions qualitatives des ports de communications

Le Zynq-7000 possède trois types de ports de communications différents en logique fixe pour assurer la communication entre le PL et le reste du système : une instance d'*Accelerator Coherency Port*, quatre *High-Performance AXI Controllers* et quatre ports *General Purpose* [16, p. 40]. Ces interfaces ont toutes leurs particularités et une performance dépendant de leur contexte d'utilisation. Une vue d'ensemble des interconnections de ces interfaces est donnée à la figure 2.6 et est détaillée plus bas. Notons que ces interfaces implémentent toutes le protocole AXI 3.0 [32].

Accelerator Coherency Port

Le PL du Zynq-7000 est maître d'un port AXI 64 bits connecté au port ACP (présenté à la section 2.3.1) du PS. Comme le montre la figure 2.4, ce port permet la communication du PL avec la mémoire interne, les périphériques du reste du SoC et la mémoire (DDR) externe, le tout en restant cohérent avec la cache des processeurs.

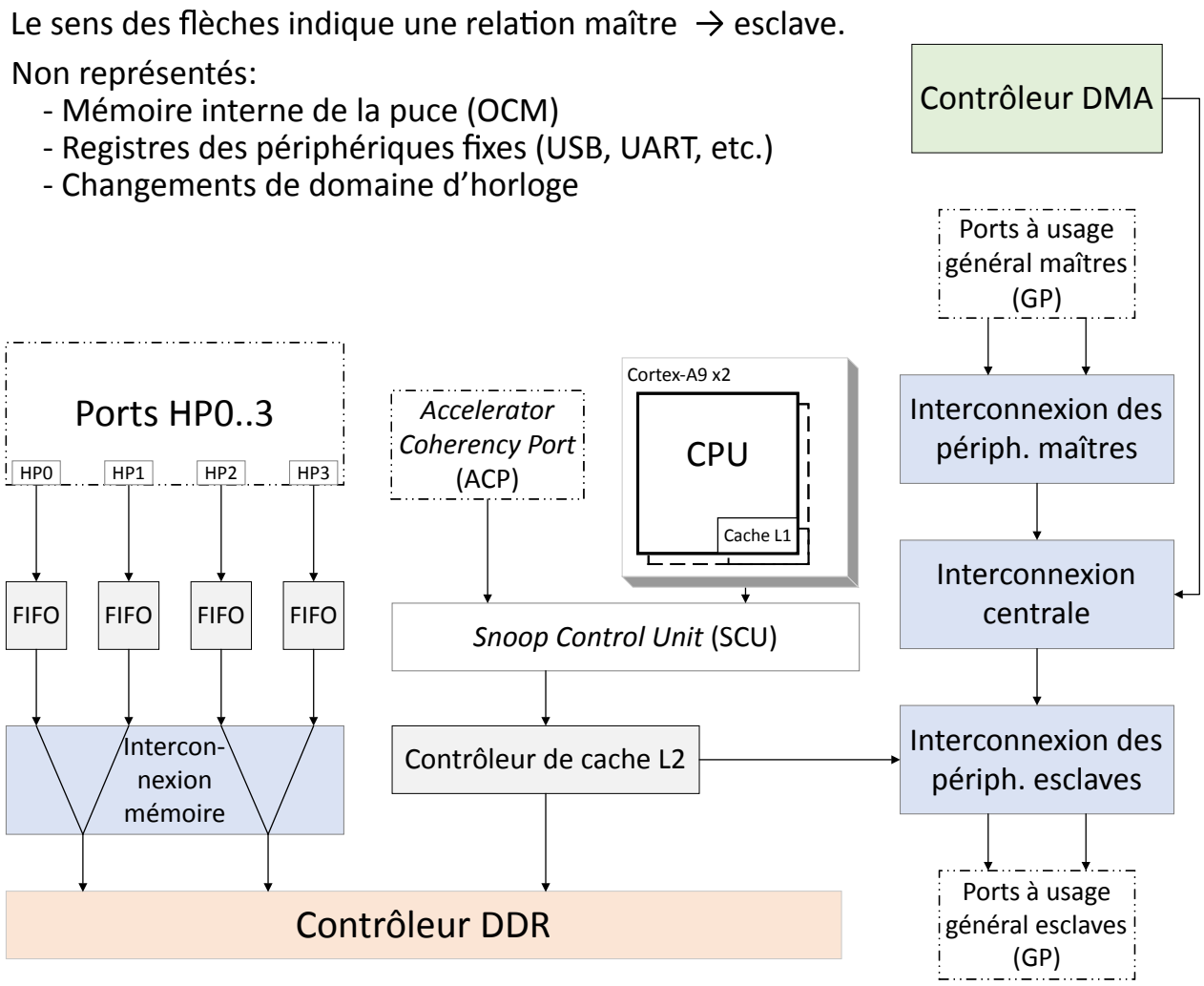


Figure 2.6 Schéma simplifié des interconnexions du Zynq-7000. Voir [16, p.120] pour plus de détails.

High-Performance AXI Controllers

Comme il est détaillé dans la partie gauche de la figure 2.6, la partie PL du Zynq-7000 est aussi le maître de 4 ports 64 bits "*High Performance*" (HP) qui permettent une connexion à la mémoire interne et à la DDR externe [16, p. 128]. Cette interface doit son nom à l'utilisation de FIFOs de 1 ko chacune qui permettent d'amortir les effets de latence de la DDR. Cependant, contrairement à l'ACP, comme la connexion ne passe pas par le SCU, en cas de partage de plages mémoire entre le PS et le PL, la cohérence de cache doit être assurée par le logiciel du PS, qui doit invalider ces plages de cache, configurer la cache en mode d'écriture immédiate (dans le cas d'un transfert PS $\rightarrow$ PL), ou configurer les pages utilisées comme ne devant pas être mises en cache.

General Purpose AXI Controllers

Finalement, le Zynq-7000 possède aussi 4 ports (2 maîtres et 2 esclaves) 32 bits à "usage général" (GP), similaires aux ports HP, mais sans la FIFO permettant un tampon efficace et étant relié à l'interconnexion des périphériques plutôt qu'à celle de la mémoire directement. Leur utilisation n'est donc pas recommandée pour une application nécessitant une bonne performance [16, p. 138]. Le détail des connexions est visible à la droite de la figure 2.6. Notons entre autres que ce n'est qu'avec les ports esclaves de cette interface qu'un coprocesseur peut être directement adressé par le processeur, ce qui en fait le moyen de configuration privilégié.

2.4.2 Prédiction de performance et facteurs affectant cette performance

La performance prévue des chemins des données du PL vers la mémoire externe est assez bien documentée, tant par Xilinx que par la littérature indépendante, ce qui permet d'avoir une bonne idée de la performance atteignable et des facteurs influençant cette performance.

Performance d'après Xilinx

Le manuel de référence du Zynq-7000 offre un estimé du débit des différents ports de communications du FPGA-SoC [16, p. 662] et est résumé dans le tableau 2.1. Il indique aussi que l'on peut s'attendre à une latence de 100 à 200 cycles du PS pour la configuration d'un accélérateur, mais que cette latence ne devrait pas être significative pour des transferts de plus de 4 ko [16, p.656]. Cependant, bien que ces chiffres donnent une bonne idée de la performance relative des moyens de communication avec le PL, de nombreux facteurs peuvent affecter négativement cette performance. De plus, l'impact de ces facteurs varie d'un port

Tableau 2.1 Estimation des performances des ports de comm. d'après Xilinx [16, p. 662]

Port de comm.	**Bande-passante maximale estimée dans une direction**	**Points marquants**
GP (CPU)	<25 Mo/s	— Suggéré pour le contrôle
GP (DMA)	600 Mo/s	— N'utilise pas de ressources du PL
HP (par port)	1200 Mo/s	— Accès à l'OCM/DDR seulement — Pas de cohérence avec la cache du PS
ACP	1200 Mo/s	— Cohérence avec la cache du PS — Partage la BP du PS vers la mémoire — Peut causer l'éviction de données utilisées par le PS

de communication à l'autre, ce qui complexifie le choix du meilleur port pour transférer les données entre le processeur et un accélérateur.

Performance selon la littérature

En regardant du côté de la littérature, ce choix, fonction de diverses caractéristiques de l'application, devient plus facile. Plusieurs auteurs [20], [21], [33], [34] ont comparé la performance des différents ports pour des transferts contigus de taille variable. Un résumé de leurs observations est présenté au tableau 2.2. Notons que l'utilisation d'un DMA pour les ports GP n'a pas été mesuré par aucun des auteurs et n'est donc pas représentée ici. La section 3.3.2 du chapitre 3 donnera un aperçu plus détaillé des points présentés dans ce tableau.

Parmi les autres contributions de la littérature, notons aussi la présence d'un modèle analytique de performance des différents chemins des données du Zynq-7000 [35]. Cependant, celui-ci offre un portrait moins complet de la situation que les données expérimentales offertes précédemment, ne modélisant notamment pas la lecture/écriture simultanée ni la combinaison de plusieurs chemins pour obtenir une meilleure performance. De plus, nos propres données expérimentales ne concordaient pas toujours avec ce modèle, possiblement en raison de la

Tableau 2.2 Estimation des performances des ports de comm. d'après [20], [21], [33], [34].

Port	Points marquants
GP (CPU)	— Une plus faible latence en fait le meilleur choix pour des transferts de 16 à 64 octets [33]. — La bande-passante plafonne sous 25 Mo/s [34].
HP	— La BP atteignable est proche de zéro pour de petits transferts et augmente jusqu'à plafonner à une taille entre 64 et 128 ko [20], [21]. — La performance est limitée par la fréquence du maître AXI connecté au port. L'augmenter augmente donc la bande-passante [21]. — On s'attend à une BP full duplex d'environ 840 Mo/s pour un accélérateur à 110 MHz [21]. — Peu affecté si le CPU fait des accès mémoires en parallèle [20].
2x HP	— Utiliser deux ports en parallèle permet d'atteindre une BP légèrement sous le double de la configuration à un seul port, si les transferts sont de taille raisonnable [21].
4x HP	— La performance est limité par le contrôleur mémoire, qui sature aux environs de 1660 Mo/s [21]. — La fréquence du PL n'a alors aucun effet significatif sur la BP [21]. — La BP obtenue est proche, mais supérieure à la BP pour la configuration 2x HP [21].
ACP	— La BP atteignable est similaire à un port HP pour des tailles inférieures à 256 ko (la moitié de la taille de la cache L2 du PS). La BP diminue significativement (jusqu'à 60%) passé cette taille [20]. — La performance est fonction de la fréquence du maître AXI connecté jusqu'à 256 ko, puis limitée par le contrôleur de cache [21]. — La performance d'un transfert de 32 octets aligné est particulièrement bonne [21]. Ceci correspond à la taille d'une ligne de cache et correspond au transfert recommandé par ARM [22]. — On note une réduction significative des performances si le CPU fait des accès mémoire en parallèle [20].

quantité insuffisante de détails des expérimentations réalisées par les auteurs pour vérifier leur modèle, qui ne nous permettaient peut-être pas de comparer des pommes avec des pommes. Pour ces deux raisons, ce modèle est ignoré dans le reste du présent mémoire.

En résumé, sur le Zynq-7000, la performance en termes de débit de données atteignable varie en fonction :

1. du port de communication choisi
2. de la quantité de données transférées
3. de la fréquence d'horloge de l'accélérateur
4. de la contiguïté (ou non) de ces données

2.4.3 Comparaison avec le SoC-FPGA Cyclone V S*x* d'Intel FPGA

Le Cyclone V SE/ST/SX d'Intel FPGA [23] est un FPGA-SoC très similaire au Zynq-7000 du point de vue de son processeur et de ses interconnexions. Il comporte un Cortex-A9 à un ou deux cœurs comme le Zynq-7000 et offre des moyens similaires d'accéder au FPGA. Cependant, plutôt que d'offrir 4 ports 64 bits vers le contrôleur de la mémoire principale comme le Zynq, le Cyclone V SE/ST/SX comporte un seul port sur lequel peut se connecter jusqu'à 3 contrôleurs AXI (ou 6 contrôleurs Avalon) d'une largeur totale ne dépassant pas 256 bits [36] (par exemple, il est possible d'avoir un port 256 bits, un port 128 bits et 2 ports 64 bits, etc.).

Côté performance, une comparaison du Cyclone V au Zynq-7020 note que le débit maximal atteignable est plus faible qu'un Zynq-7020, mais que tant que la largeur de transfert est de moins de 256 bits, la performance reste limitée par la fréquence du FPGA, de la même manière que le Zynq-7000 [21].

2.4.4 Comparaison avec un contrôleur mémoire sur FPGA et un contrôleur Microblaze

Gobel et al. [21] ont aussi comparé l'utilisation du processeur ARM et de son contrôleur mémoire fixe à un processeur Microblaze et un contrôleur mémoire en logique programmable (connecté à une mémoire externe différente) sur un Zynq-7045. Cette deuxième option permettant de configurer la largeur du bus mémoire à 512 bits (plutôt que 4x64), les auteurs obtiennent un débit nettement supérieur à la première option, à plus de 8,5 Go/s en lecture ou en écriture. Ceci confirme qu'utiliser un contrôleur mémoire fixe mais conçu pour un processeur général peut limiter la performance atteignable de l'accélérateur sur FPGA.

2.5 Synthèse de haut-niveau

La synthèse de haut-niveau (HLS) est le processus de transformation d'une représentation comportementale et abstraite d'un circuit électronique vers une représentation logique (au niveau RTL) équivalente. Cette deuxième représentation peut par la suite être fournie en entrée à un synthétiseur qui la transforme alors en flux binaire (*bitstream*) prêt à être exécuté sur un FPGA. On peut séparer les outils HLS en deux grandes catégories, les premiers prenant en entrée un langage conçu sur mesure pour augmenter le niveau d'abstraction par rapport aux HDLs traditionnels (VHDL, Verilog) tout en gardant une représentation assez proche du matériel alors que les deuxièmes prennent en entrée des langages généraux pré-existants, permettant la réutilisation de code logiciel conçu pour un processeur général. Cette première catégorie comporte des outils comme Bluespec SystemVerilog [37] ou Chisel [38], qui apporte une approche de programmation fonctionnelle à la conception de circuits électroniques. Les outils de la deuxième catégorie ciblent en prédominance le C/C++, mais aussi OpenCL [39] ou Matlab [40]. Cette deuxième catégorie (et plus précisément les outils ciblant le C/C++) étant la cible de ce travail, c'est à celle-ci que nous référerons comme outils de synthèse de haut-niveau dans le reste du document.

Côté performance et utilisation de ressources, la première catégorie d'outils offre en général un résultat très proche d'une version décrite directement en HDL [41], mais la deuxième offre des résultats plus mitigés, pouvant donner des résultats médiocres [41] ou comparables bien qu'inférieurs à l'HDL [42]. De manière générale, la littérature et nos expériences personnelles dénotent qu'il est nécessaire d'appliquer des modifications substantielles à un code provenant d'une implémentation logicielle avant de le synthétiser, notamment en changeant sa structure pour assurer de meilleurs accès mémoire, en inférant une mémoire *scatchpad*, en ajoutant des directives (généralement sous la formes de *pragmas* dans le code) pour guider le synthétiseur, etc. Il n'est pas rare de voir l'ensemble de ces modifications apporter une différence de performance de plus de 100×.

2.5.1 Deux outils HLS modernes

La liste des outils ciblant le C/C++ étant longue, nous ne détaillerons ici que Vivado HLS et LegUp mais des revues récentes et plus exhaustives sont disponibles à [43] et [44].

Vivado HLS

Vivado HLS [1] (anciennement AutoESL) est un outil de synthèse de haut niveau développé par Xilinx pour les FPGA Xilinx. Il prend en entrée du code C/C++/SystemC/OpenCL,

optionnellement agrémenté de directives, et produit un RTL en Verilog ou VHDL. Bien que limité aux FPGA de Xilinx, Vivado HLS est l'un des outils HLS les plus utilisés dû à son coût originalement négligeable (et aujourd'hui gratuit pour certains FPGA) par rapport aux autres outils disponibles sur le marché. L'outil est basé sur une version fortement modifiée de LLVM 3.1 (vu à la section 2.6.2).

Outre des simulateurs C et RTL et un débogueur, l'outil permet une estimation des ressources utilisées et de la latence d'exécution du module synthétisé. Cependant, comme Vivado HLS n'a pas de vue d'ensemble du système complet, il modélise mal la latence de communications vers une mémoire externe. Il est donc difficile d'estimer avec précision le temps d'exécution d'un module, en comptant ses accès vers une mémoire externe, uniquement avec Vivado HLS.

LegUp

Originalement un logiciel libre et gratuit pour des fins de recherche, LegUp [45] est un logiciel permettant de transformer du code C en Verilog. Les deux particularités les plus notables de cet outil sont le support de tous les fabricants de FPGA ainsi que la possibilité d'utiliser directement les librairies *pthread* et OpenMP pour inférer un parallélisme matériel, de la même manière qu'en logiciel. Comme Vivado HLS, LegUp est basé sur LLVM.

2.5.2 Environnements de développement au niveau système

L'un des plus grands avantages de partager le même langage de programmation entre un accélérateur matériel et un programme logiciel est la possibilité de simplification d'un flot de développement au niveau système, où la configuration et le design d'un SoC se fait en même temps que sa programmation. Communément appelée codesign logiciel/matériel, cette approche permet par exemple, à l'aide d'outils supportant ce flot, de faire passer une fonction exécutée sur un processeur général vers une implémentation matérielle (ou vice versa) à l'aide d'un seul clic de bouton puis d'évaluer la performance de cette nouvelle configuration. C'est également un excellent moyen d'évaluer la performance globale du système développé, en tenant compte de toutes les interactions entre les différentes modules, des communications avec la mémoire principale, etc., ce qui n'est pas possible en évaluant chacun de ceux-ci séparément.

SpaceStudio

SpaceStudio [46] est un logiciel d'exploration architecturale reposant sur le concept de *plateforme virtuelle*, soit une plateforme de simulation permettant de reproduire le fonctionne-

ment d'un SoC complet. Il permet, pour un système décrit en C/C++ et séparé en modules communiquant par un API spécifique, de mapper chacun de ces modules et moyens de communications à des ressources matérielles spécifiques, puis de simuler le système complet. Par exemple, il permet d'évaluer rapidement la performance obtenue si un certain module est implémenté comme processus logiciel exécuté sur un processeur ARM multicœur, comme matériel sur FPGA, comme tâche temps réel sur un processeur donné (ARM, Microblaze, Nios, etc.). ou si telle configuration de bus est utilisée plutôt qu'une autre pour acheminer les communications entre différents modules.

SpaceStudio délègue la synthèse des modules matériels à des outils de synthèse de haut-niveau tiers et permet l'exportation du système développé vers une forme implémentable sur FPGA. Il supporte les FPGA de Xilinx et d'Intel FPGA.

SDSoC™

SDSoC™ [47] est un environnement de développement (IDE) basé sur Eclipse et fourni par Xilinx pour les FPGA-SoC et FPGA de la compagnie. Celui-ci permet de bâtir un système logiciel comportant des parties accélérées sur FPGA tout en conservant un flot de développement similaire à un flot de développement C/C++ standard.

Pour développer une implémentation matérielle d'une fonction du code, il suffit de l'indiquer à l'outil et optionnellement d'identifier l'interface de communications à utiliser, si le défaut ne convient pas. L'outil synthétisera alors cette fonction avec Vivado HLS et générera le contrôle des communications en analysant les endroits où cette fonction est appelée. Il permet la génération d'un *bitstream* pour exécution sur le FPGA, le débogage à distance et une estimation rapide des performances atteignables, basée sur une combinaison du temps d'exécution de la fonction (fournie par Vivado HLS) et du moyen de communication choisi. Cependant, pour ce dernier point, l'analyse effectuée se limite présentement à regarder la taille des éléments transférés par l'accélérateur et le moyen de communication choisi. Plus précisément, SDSoC ne regarde pas la manière dont le transfert est fait par l'accélérateur (par exemple, la présence ou non de transferts en rafale), ce qui peut causer dans certains cas des différences significatives entre l'estimation et la réalité.

Environnement de développement de LegUp

De manière similaire à SDSoC, LegUp offre la possibilité de générer un système comportant processeur ARM et logique programmable, en indiquant quelles fonctions d'un projet logiciel doivent être exécutées en matériel [48].

2.5.3 Suite de bancs de test *CHStone*

CHStone [49] est une suite de 12 bancs de test écrite en C et développée spécialement pour comparer la performance de différents outils de synthèse de haut-niveau. Ces différents bancs de test sont en majeure partie extraits d'autres suites de tests ou sont des logiciels destinés à des processeurs plus généraux et modifiés pour être synthétisables facilement. On y retrouve notamment des algorithmes de cryptographie, un décodeur JPEG, un décodeur de vecteur de mouvement, un processeur MIPS simplifié, etc. Ces bancs de test ont en moyenne environ 750 lignes de code chacun et 7 d'entre eux font des opérations sur des vecteurs, alors que les 5 autres opèrent uniquement sur des scalaires.

Les bancs de test vectoriels de cette suite furent utilisés pour les premiers tests de ce présent projet car ils représentent des cas assez simples (ex. synthétisables sans modifications, avec des tailles de tableaux et itérations de boucles constantes, etc.) mais assez représentatifs d'accélérateurs réels.

2.6 Analyse statique de code source

L'analyse *statique* d'un programme est une analyse de son code source seulement, sans exécution du programme, à l'opposé d'une analyse dite *dynamique*. Cette analyse est utile pour déterminer l'ensemble des options de contrôle et de flot de données d'un programme, pour effectuer des transformations de code automatisées selon des conditions spécifiques, etc. Bien que l'analyse statique ne permette pas de connaître la valeur de la majorité des variables du programme (qui ne sont connues qu'à l'exécution), elle permet souvent de raisonner sur l'ensemble des valeurs qu'elles peuvent prendre, contrairement à l'analyse dynamique qui n'offre pas de telles garanties.

Nous verrons ici trois outils permettant d'effectuer de telles analyses sur du code C/C++ : Clang et LLVM, principalement utilisés pour la compilation mais étant intentionnellement développées pour faciliter d'autres possibilités d'utilisations ainsi ROSE, un projet de recherche développé pour permettre de telles analyses.

Les deux premiers outils font partie du *projet LLVM* [50]-[52], qui est une collection d'outils pour compilateurs. LLVM est un optimiseur et générateur de code machine prenant en entrée du code sous la forme d'une représentation intermédiaire (IR) alors que Clang est principalement un compilateur C/C++ ciblant cette représentation intermédiaire. Les descriptions de Clang & LLVM présentés à la section 2.6.1 et 2.6.2 s'appliquent à la version 4.0 de LLVM et Clang (datant de mars 2017).

2.6.1 Clang

Clang [53] est une partie frontale pour LLVM permettant notamment la traduction de code C, C++, OpenCL et Objective C en représentation intermédiaire LLVM et avec une interface compatible avec GCC ou MSVC. Outre cette utilisation destinée à un flot de compilation standard, Clang est, comme LLVM, conçu sous forme de librairie C++ moderne ayant pour but de faciliter le développement d'outils connexes.

Cette approche de librairie, permet à Clang d'être utilisé :

— Pour le réusinage systématique de grandes quantités de code [54].

— Pour la détection de bogues par analyse statique [55].

— Comme analyseur de syntaxe d'IDEs [56].

Comme plusieurs compilateurs, Clang représente à l'interne le résultat de son analyse syntaxique du code sous forme d'un arbre de syntaxe abstrait (AST). Cependant, à l'inverse d'autres compilateurs comme GCC, l'AST de Clang conserve la totalité des informations nécessaires pour reconstruire la source, permettant à un utilisateur de parcourir l'AST et le modifier puis de produire le code source modifié correspondant. Ceci facilite aussi l'affichage d'avertissements ou d'erreurs lors du parcours de l'AST en permettant d'indiquer l'endroit exact correspondant au nœud concerné dans le code source.

Transformations et analyse de source avec Clang

Clang expose ses fonctionnalités internes par le biais de la librairie *LibTooling*. Cette librairie permet notamment de créer des instances de la classe *ClangTool*, qui sert à exécuter des *FrontendActions* (actions sur le code source) sur des fichiers en fonction d'arguments de compilation passés en ligne de commande. Elle permet aussi de séparer facilement les arguments de cette ligne de commande destinés à Clang de ceux destinés à l'outil développé.

Le *FrontendAction MatchFinder* trouve les instances d'un patron précis de nœuds de l'AST lors du parcours de celui-ci et exécute une fonction de rappel associée (*MatchCallback*) lorsque ce patron est rencontré. Les figures 2.7a à 2.7c montrent des exemples de création de tels patrons. Les figures 2.7a et 2.7b montrent par exemple comment être informé de toutes les déclarations de paramètres de fonctions et comment obtenir un pointeur vers le noeud de l'AST correspondant à la fonction dont le paramètre fait partie. Cependant, comme le montre la figure 2.7c, il est aussi possible de spécifier des patrons beaucoup plus complexes. On peut aussi identifier les nœuds jugés intéressants, ce qui facilite leur accès une fois dans la fonction

```
parmVarDecl(           // On cherche les déclarations de paramètres
).bind("paramètre")    // On préserve un pointeur vers cette déclaration
```

(a) Patron pour trouver toutes les déclarations de paramètres de fonctions.

```
parmVarDecl(
    hasAncestor(functionDecl().bind("fonction"))
).bind("paramètre")
```

(b) On préserve en plus un pointeur vers la fonction concernée.

```
declRefExpr(    // On cherche les utilisations (références) de déclarations
    allOf(
        anyOf(
            // Si cette utilisation fait partie d'une déclaration de variable
            hasAncestor(varDecl().bind("déclaration de variable")),
            // ou si elle fait partie d'une opération (ex. a + b)
            hasAncestor(binaryOperator().bind("opération"))
        ),
        to(parmVarDecl(    // si la déclaration concerne un paramètre
            hasAncestor(functionDecl().bind("fonction"))
        ).bind("paramètre")),
        unless(                         // sauf s'il s'agit de l'utilisation
            hasParent(memberExpr())     // d'un membre d'une struct. ou classe
        )
    )
).bind("utilisation")
```

(c) On cherche toutes les utilisations des paramètres d'une fonction.

Figure 2.7 Exemples de différents patrons pouvant être fournis à un *MatchFinder*

de rappel. Il est ensuite possible d'accéder au nom de cette fonction, à sa position dans le code, etc.

De plus, les applications de réusinage de code étant parmi les principaux utilisateurs de *LibTooling*, Clang offre également la classe *RefactoringTool*, une spécialisation de *ClangTool*. Celle-ci ajoute le concept de liste de remplacements de portions de code, qui est remplie lors du parcours de l'AST. Cette liste de remplacements peut ensuite être appliquée automatiquement à la source originale.

2.6.2 LLVM

Bien qu'étant principalement un optimiseur et un générateur de code, il est possible d'utiliser LLVM pour analyser statiquement du code de deux manières différentes : 1) en l'utilisant comme librairie ou 2) en écrivant un *plugin* chargé dynamiquement par l'exécutable standard de l'optimiseur. Ces deux méthodes donnent accès à toutes les analyses utilisées par le processus d'optimisation, comme le graphe de flot de contrôle, le nombre d'itérations des différentes boucles, une estimation des probabilités de branchements, les dépendances, etc.

Le reste de cette sous-section décrit différents points techniques du fonctionnement de LLVM nécessaires à la compréhension du chapitre 4.

Représentation intermédiaire LLVM

L'IR LLVM [57] est un langage à mi-chemin entre l'assembleur et un langage de haut niveau. Il a pour but de pouvoir représenter n'importe lequel de ces derniers tout en étant assez simple pour faciliter les optimisations, notamment en étant sous la forme *Static Single Assignment* (SSA), c'est-à-dire sous une forme où chaque variable est assignée exactement une seule fois. Un exemple de l'IR utilisant cette forme est montré à la figure 2.8. Notons également que les instructions sont typées, incluant les notions de taille (en bits), de pointeur, de structures et de constantes. De plus, le code peut être séparé en fonctions. Finalement, notons qu'en raison de l'assignation unique de la forme SSA, une instruction est son résultat au niveau de la hiérarchie des classes du langage.

Les instructions d'une fonction y sont séparées en blocs de base (BB), qui contiennent chacun une suite d'instructions qui est garantie de s'exécuter consécutivement sans branchement depuis ou vers n'importe quel portion du code, sauf pour la première instruction (branchements depuis le reste du code) et la dernière (branchements vers le reste du code). Cette séparation fait de chaque bloc de base un vertex dans le graphe de flot de contrôle du programme, facilitant certaines optimisations et analyses. La figure 2.9 présente une boucle simple en C

```c
int SSA(int a, int b)
{
    int z = (a + b) * b;
    return z;
}
```

(a) Code C

```
define i32 @SSA(i32 %a, i32 %b) {
    %0 = add i32 %a, %b
    %1 = mul i32 %0, %b
    ret i32 %1
}
```

(b) Équivalent SSA en représentation LLVM

Figure 2.8 Code simple et son équivalent SSA en IR LLVM.

et en IR LLVM pour montrer cette séparation en blocs. On y note 2 blocs principaux, *BB1* qui calcul l'index de boucle et *BB2* qui exécute le contenu de celle-ci. Notons également l'utilisation à la ligne 9 de l'instruction *phi*, qui assigne une valeur selon le bloc précédemment exécuté : cette façon de faire est due à la forme SSA, qui empêche d'initialiser une variable dans un bloc (*BB0* ici) et de la modifier dans un autre (*BB2*).

Instruction GetElementPtr

L'IR LLVM possède l'instruction dédiée *GetElementPtr* (GEP) pour calculer l'adresse d'un élément d'un tableau ou structure, plutôt que d'utiliser explicitement des instructions de multiplications et addition. Ceci permet de séparer les calculs d'adresses d'éléments des autres calculs arithmétiques, ce qui simplifie le travail des analyses de dépendances mémoire, d'élimination d'accès mémoire identiques, etc. Cette instruction est d'une importance critique pour l'analyse de contiguïté d'accès mémoire, puisque c'est le résultat de cette instruction qui change au fur et à mesure de l'exécution d'une boucle. L'instruction prend la forme :

```
%resulat = getelementptr type, type* %addresseDeBase, [%index1, .. %indexN]
```

où le nombre d'indices correspond au nombre de dimensions du type de donnée. Par exemple, le calcul de l'adresse de l'accès *a[i]* correspond à :

```
%0 = getelementptr i32, i32* %a, i32 %i
```

L'instruction GEP ne fait aucun accès mémoire : elle ne fait que calculer une adresse. En règle générale, le résultat de l'instruction sera par la suite utilisé par une instruction *load* ou *store* mais il pourrait aussi être utilisé dans une comparaison de pointeurs, dans quel cas aucun accès mémoire n'est fait, même par les utilisateurs de l'instruction. Notons aussi que

```c
void loop(int* a, int* b)
{
    for (int i = 0; i < 42; ++i)
        a[i] = b[i];
}
```

(a) Code C

```
define void @loop(i32* %a, i32* %b) {
BB0:
  ; On saute inconditionnelement à BB1
  br label %BB1

BB1:                                              ; prédécesseurs = %BB2, %BB0
  ; On assigne le compteur de boucle à 0 ou indvars.iv.next
  ; selon le bloc précédent
  %indvars.iv = phi i64 [ %indvars.iv.next, %BB2 ], [ 0, %BB0 ]
  ; On compare le compteur de boucle à la condition de sortie (42)
  %exitcond = icmp ne i64 %indvars.iv, 42
  ; On saute à BB2 ou BB3 selon exitcond
  br i1 %exitcond, label %BB2, label %BB3

BB2:                                              ; prédécesseur = %BB1
  ; On détermine le pointeur vers b[indvars.iv]
  %0 = getelementptr inbounds i32, i32* %b, i64 %indvars.iv
  ; On charge la donnée
  %1 = load i32, i32* %0, align 4
  ; On détermine le pointeur vers a[indvars.iv]
  %2 = getelementptr inbounds i32, i32* %a, i64 %indvars.iv
  ; On écrit la donnée
  store i32 %1, i32* %2, align 4
  ; On calcul la prochaine valeur de la variable d'induction (i)
  %indvars.iv.next = add i64 %indvars.iv, 1
  ; On saute inconditionnelement à BB1
  br label %BB1

BB3:                                              ; prédécesseur = %BB1
  ret void
}
```

(b) Équivalent en IR LLVM et séparation en blocs de base

Figure 2.9 Code simple avec branchements et sa séparation en blocs de base.

le même GEP peut avoir plus d'un utilisateur. Par exemple, dans le cas d'une accumulation, où il faut charger une donnée puis la réécrire modifiée, le même GEP serait utilisé par une instruction *load* et une instruction *store*.

Métadonnées et information de débogage

L'IR LLVM permet aussi d'attacher des métadonnées aux instructions. Ceci sert à transmettre des informations supplémentaires depuis l'application frontale, incluant des informations sur la source originale pour le débogage ou, dans certains cas, pour transmettre des informations d'une passe à l'autre. Les passes d'optimisation sont chargées de conserver le plus fidèlement possible ces informations lorsqu'elles modifient les instructions du programme. Dans le cas d'informations de débogage, cela permet d'afficher un avertissement ou une information à un utilisateur en référençant correctement l'endroit concerné dans le code source.

Passes LLVM

L'analyse, l'optimisation et la transformation en code machine réalisée par LLVM depuis la représentation intermédiaire (IR) fournie en entrée se fait par le biais de l'application d'une succession de *passes*. Ces passes font ou bien une analyse du code fourni en entrée ou appliquent des modifications sur celui-ci. Si une passe modifie le code, c'est cette version modifiée qui est alors fournie en entrée aux passes suivantes. Une passe peut dépendre du résultat de certaines analyses, mais pas de certaines optimisations.

Un gestionnaire de passes est chargé d'exécuter toutes les passes d'optimisation dans l'ordre demandé, tout en exécutant les passes d'analyse nécessaires entre ces passes. Il ré-exécute notamment les analyses nécessaires si des passes précédentes les ont invalidées en modifiant l'IR. La modification de l'IR suite à une passe d'optimisation peut ouvrir de nouvelles possibilités d'optimisation à d'autre passes. Il est donc courant que certaines passes d'optimisations soient exécutés plusieurs fois.

De plus, dans un soucis de rapidité d'exécution, chaque passe ne fait que le strict minimum nécessaire, ce qui fait en sorte que, dépendamment de l'IR en entrée, il faille parfois appliquer certaines passes d'optimisations avant que certaines passes d'analyse ne donnent des résultats significatifs. Par exemple, la passe d'analyse donnant les informations sur le nombre d'itérations de boucles n'analyse que les registres pour déterminer la variable d'induction de cette boucle. Cependant, l'IR généré par Clang stocke et charge cette variable en mémoire. Par conséquent, sans exécuter préalablement la passe de transformation d'accès mémoire en

registres, l'analyse du nombre d'itérations de boucle retournera comme information que le nombre d'itérations est inconnu. Ce comportement est cependant assez bien documenté [58].

Passe d'évolution scalaire

La passe d'évolution scalaire (*ScalarEvolution*) est la passe LLVM qui rend possible l'analyse de la contiguïté de transactions mémoires effectuées successivement dans une boucle. Cette passe évalue l'évolution du résultat de chaque valeur entière ou pointeur au cours de l'exécution du programme en utilisant des chaînes de récurrences [59]. Ceci permet de déterminer l'évolution des variables d'une itération de boucle à l'autre et de déterminer le nombre d'itérations de boucles.

Le résultat de cette analyse est un objet dérivé de la classe de base **SCEV** pour chaque instruction analysée. Les classes dérivées possibles consistent en :

- **SCEVConstant** : la valeur est une constante.
- **SCEVAddExpr**, **SCEVMulExpr**, **SCEVUDivExpr** : la valeur est le résultat d'une addition, multiplication ou division entière, respectivement.
- **SCEVAddRecExpr** : la valeur est incrémentée d'une autre valeur à chaque itération d'une boucle spécifiée.
- **SCEVUnknown** : il n'est pas possible d'évaluer l'évolution de la valeur.
- **SCEVCouldNotCompute** : l'analyse n'est pas capable de déterminer l'évolution de la valeur.
- Quelques opérandes de *cast* et d'extension de types (par exemple, conversion d'une représentation 32 bits vers 64 bits) qui sont de faible importance pour ce travail.

Les opérandes de tous les **SCEV** sont eux-même des **SCEV**, sauf pour **SCEVConstant** (où l'opérande est une constante), **SCEVUnknown** (où l'opérande est la variable n'ayant pas pu être évaluée) et **SCEVCouldNotCompute** (où il n'y a pas d'opérande). La récursivité est exprimée en ayant une opérande d'un type autre que ces trois types. Par exemple, l'opérande d'un **SCEVAddExpr** pourrait être un autre **SCEVAddExpr**, qui pourrait avoir lui-même comme opérande un **SCEVUDivExpr**, etc. De plus, les **SCEV** de type **SCEVAddExpr**, **SCEVMulExpr**, et **SCEVAddRecExpr** sont des opérateurs n-aires (c'est-à-dire qu'un **SCEVAddExpr** pourrait représenter la somme de 7 opérandes, par exemple).

Un exemple de résultats de l'analyse d'évolution scalaire effectuée sur les instructions GEP d'une fonction est présenté à la figure 2.10. Pour favoriser la brièveté, les **SCEVConstant** et

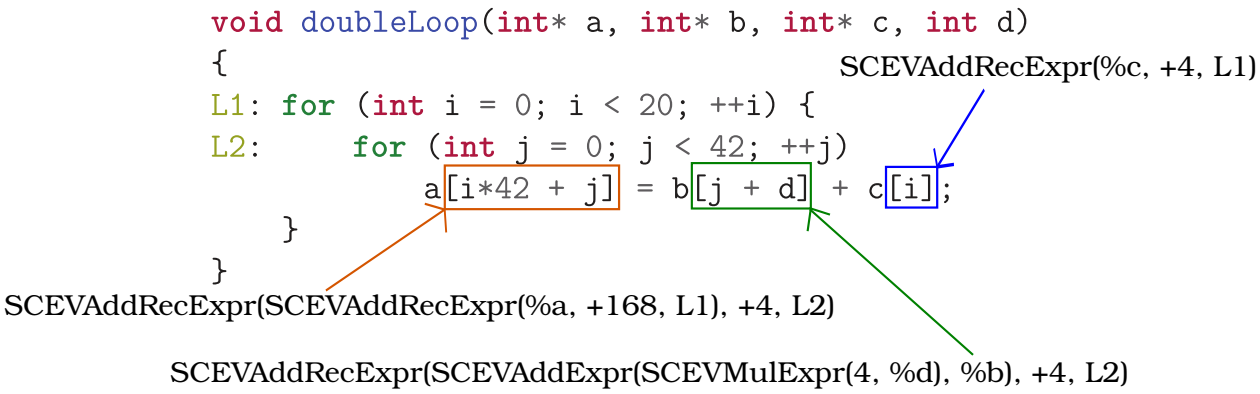


Figure 2.10 Exemple de résultats obtenus par l'analyse d'évolution scalaire.

les **SCEVUnknown** y sont représentés sous la forme de la constante et la variable qu'ils encapsulent, respectivement. De plus, la nomenclature pour détailler un **SCEVAddRecExpr** est sous la forme *valeur à l'itération 0*, *addition par itération de boucle*, *boucle concernée*.

L'analyse d'évolution scalaire ne considère que les instructions directement modifiées par une variable d'induction de boucle (qu'elle soit explicite ou non). Ainsi, l'exemple de la figure 2.10 traite de l'évolution de la valeur des instructions GEP menant subséquemment à une instruction *load* (pour b et c) ou *store* (pour a), car ce sont les GEP qui utilisent directement les variables d'induction des boucles dans leurs opérandes. Du point de vue de l'analyse d'évolution scalaire, les instructions *load* et *store* ne varient pas dans le temps de manière analysable : elles ont toujours comme opérande le résultat du GEP correspondant.

Passe d'informations de boucles

Étant un langage proche de l'assembleur, l'IR LLVM ne possède pas en soi de notation particulière pour exprimer des boucles et est limité à représenter des branchements d'un bloc de base à un autre. La passe d'analyse d'informations de boucle sert donc à relever le niveau d'abstraction en détectant les boucles naturelles (qui ne possèdent qu'un seul bloc d'entrée) d'un programme, en identifiant les sous-boucles, en identifiant le ou les blocs de sortie de boucles, en permettant de déterminer à quel boucle appartient un bloc de base donné, etc.

Cette passe est pré-requise à la passe d'analyse d'évolution scalaire, qui peut alors s'en servir pour donner des informations plus poussées sur les boucles comme le nombre maximal et minimal d'itérations depuis chaque sortie possible de chaque boucle.

Passe de probabilité de branchement

La passe d'analyse de probabilité de branchement (*BranchProbabilityInfo*) estime la probabilité qu'un noeud *source* du graphe de flot de contrôle (c'est-à-dire un bloc de base) passe directement à un noeud *destination* plutôt qu'à un autre noeud. Le but de cette analyse, dans un flot de compilation standard, est de déterminer quels noeuds du CFG ont une forte chance de s'exécuter pour guider les optimisations de localité d'instructions.

L'exactitude de l'estimation n'est pas primordiale pour un flot de compilation régulier, puisqu'elle ne sert qu'à déterminer si un branchement a plus de chances de se produire qu'un autre. Ainsi, les développeurs de LLVM ont préféré la simplicité et la rapidité d'exécution, en retournant simplement une probabilité donnée par des poids pré-calculés. Par exemple, la probabilité de branchement estimée par l'analyse pour une condition booléenne sera toujours de 50%. Il s'agit du même résultat si un entier est comparé à une valeur arbitraire, sauf si cette valeur arbitraire est 0, dans quel cas la probabilité retournée est de 37,5%. De même, la probabilité estimée qu'une boucle effectue une autre itération est toujours de 96,88%, plutôt que d'être basée par exemple sur le nombre d'itérations de boucles prévues.

2.6.3 ROSE

ROSE [60] est une infrastructure pour compilateur ayant pour but de faciliter l'analyse et les transformations source-à-source de langages C, C++ et Fortran. Le projet fournit des moyens de transformer le code source vers un AST, de le traverser en y apportant des modifications et de réécrire un code source résultant des transformations. Existant depuis 2000, ROSE a notamment été utilisé pour faire de l'analyse de boucles, pour traduire des directives OpenMP, évaluer la couverture de code, etc.

2.7 Améliorations du résultat de la synthèse de haut-niveau par transformations source-à-source automatisées

Tel que montré à la section 2.5, la synthèse de haut-niveau de langages logiciels comme le C/C++ peut donner des résultats proches de ce qu'une implémentation HDL manuelle ferait, mais seulement si une attention particulière est portée à la conception d'une implémentation efficace. Il faut par exemple s'assurer que le code passé à l'outil HLS est pipelinable, gère sa propre cache, effectue le maximum de transferts de données en rafale, etc. L'amélioration automatique de la qualité de ces résultats est donc un sujet de recherche d'actualité et celle-ci est généralement faite sous la forme de transformations source-à-source (ex. C++ vers C++), dont le résultat est ensuite passé à l'outil HLS pour synthèse. Cette approche est préférée à

une intégration directe à un outil HLS car ceux-ci ne sont pas standardisés, le code source de la majorités d'entre eux est propriétaire et cela permet de recibler plus facilement ces transformations d'un outil à l'autre.

2.7.1 Amélioration automatisée de la hiérarchie de mémoire

Les outils HLS utilisent la BRAM des FPGA pour stocker les données de tableaux locaux. Or, ceux-ci comportent un nombre limité de ports d'accès (par ex. 2 chez Xilinx), ce qui limite les possibilités de parallélisme s'il n'y a pas assez de ports pour tous les accès qui pourraient être effectuées dans un même cycle. Une solution courante à ce problème est de diviser les tableaux dans plusieurs unités BRAM différentes, augmentant le nombre de ports disponibles au détriment de la quantité de ressources utilisées. Cependant, cela ne règle le problème que si chaque donnée nécessaire est toujours dans une BRAM différente lors de l'exécution du coprocesseur. Ciladro et Gallo [61] présentent une manière de séparer automatiquement les données en fonction de leur utilisation, ressortant un partitionnement optimisé pour limiter les conflits d'accès.

Pouchet et al. [62] suggèrent également d'utiliser le modèle du polyèdre [63] pour appliquer une suite de transformations aux boucles affines d'un programme dans le but de déterminer les accès mémoires réutilisés d'une itération à l'autre de ces boucles. Ceci permet de générer automatiquement des caches en BRAM pour ces données et de modifier la forme de la boucle de manière à minimiser la taille de ces caches. Les résultats présentés sont prometteurs, avec des performances de 10 à 500× supérieures au cas sans cache et similaires à ce qu'il est possible d'obtenir avec une implémentation améliorée à la main.

Finalement, le projet Scavenger [64] permet la génération automatisée et personnalisée d'une cache de niveau 2 sur le FPGA avec les BRAM non-utilisés par le reste du système.

2.7.2 Réorganisation automatique de boucles

L'un des avantages principaux de l'implémentation matérielle d'un algorithme est de pouvoir pipeliner ou dérouler des boucles pour effectuer plusieurs opérations en parallèle, mais ceci n'est possible que s'il n'y a pas de dépendances de données entre les itérations. Dans certains cas, ces dépendances de données sont non uniformes d'une itération à l'autre ou séparés par un certain nombre d'itérations. Il est alors possible de séparer automatiquement ces boucles pour isoler les itérations posant des problèmes de dépendances de données de celles qui peuvent être exécutées de manière parallèle [65]. De plus, si la présence ou non de dépendances d'une itération à l'autre ne peut être déterminé statiquement parce qu'elle dépend de la valeur d'une

variable à l'exécution, Alle et al. [66] suggèrent, via une transformation source-à-source, de générer deux versions de l'accélérateur : une première, plus lente, qui suppose une dépendance de donnée et la deuxième, plus rapide, qui suppose qu'il n'y en a pas. La version à choisir est ensuite déterminée à l'exécution.

Finalement, une approche similaire à celle de Pouchet et al. présentée plus haut peut aussi être employée pour modifier l'ordre d'itérations de boucles successives ayant des dépendances de données partielles entre elles dans le but de démarrer l'exécution de la deuxième boucle parallèlement à la première [67].

2.7.3 Synthèse efficace de structures dynamiques

Les outils HLS supportent mal les structures contenant des pointeurs vers d'autres structures. Bien qu'ils puissent les synthétiser sous certaines conditions, ils n'ont pas les informations nécessaires pour paralléliser efficacement les accès mémoires de ces pointeurs. Winterstein [68] utilise la logique de sépration pour séparer les données allouées dynamiquement en ensembles disjoints ayant chacuns leurs caches privées indépendantes, permettant un accès parallèle à chacune de ces caches.

2.7.4 Compilateur Merlin

Le compilateur Merlin [69] est une solution commerciale ayant pour but, via des transformations source-à-source, de transformer automatiquement du code C/C++ logiciel vers une forme donnant une bonne performance si synthétisé par un outil HLS. Ciblant des applications dans les centres de données, l'outil promet des réorganisation de boucles pour limiter les dépendances, la génération automatiques de caches si des données sont réutilisées, une synchronisation efficace entre tâches, le déroulage efface de boucles, etc.

2.8 Résumé

Ce chapitre servait d'abord à mettre en évidence l'importance de tenir compte du débit et de la latence d'accès vers la mémoire très tôt lors de la réalisation d'un coprocesseur sur FPGA cherchant à atteindre une bonne performance. Nous avons vu qu'il peut être impossible d'améliorer la performance d'un coprocesseur par rapport à un processeur à usage général si l'intensité arithmétique de l'algorithme à accélérer est faible. Le débit est alors le goulot d'étranglement : il faut donc plutôt chercher à améliorer celle-ci pour obtenir une meilleure performance. Nous avons aussi vu que plusieurs facteurs affectent cette performance, comme la présence ou non de transactions en mode rafale, la quantité de données transférées et,

dans des cas comme les FPGA-SoC, le chemin des données utilisé. Ainsi, même s'il est critique d'évaluer le débit atteignable très tôt dans le développement d'un coprocesseur, la modélisation de celle-ci n'est pas triviale et dépend à la fois du matériel utilisé et de caractéristiques de l'algorithme à accélérer.

De plus, nous avons vu qu'il est possible de réaliser un coprocesseur sur FPGA ou ASIC à partir de code C/C++ directement. Par contre, des modifications doivent généralement être apportées à ce code pour obtenir des performances comparables à ce qui est atteignable avec des HDL. Ces modifications incluent, mais ne sont pas limitées à des modifications cherchant à améliorer la gestion de la mémoire, en favorisant les transactions en rafales et en développant une cache personnalisée pour réutiliser les données.

Dû à sa plus grande utilisation, les outils permettant d'analyser du code écrit en C/C++ plutôt qu'en HDL sont beaucoup plus poussés. Notamment, des compilateurs C/C++ à code source ouvert permettent l'analyse de code en étant utilisés comme librairies, offrant à un utilisateur tout ce qui est utilisé par le compilateur lui-même pour transformer et optimiser le code source en représentation machine. Cette approche est entre autres déjà utilisée pour tenter d'améliorer automatiquement la performance de code C/C++ destiné à être synthétisé en coprocesseur matériel.

CHAPITRE 3 CARACTÉRISATION DE LA PERFORMANCE DES COMMUNICATIONS D'UN FPGA-SoC

La section 2.2 du chapitre précédent montrait que la bande-passante vers la mémoire est souvent le goulot d'étranglement de la performance d'un coprocesseur sur FPGA. Il est donc primordial de pouvoir estimer celle-ci pour pouvoir estimer correctement la latence d'exécution totale de ce coprocesseur. Or, comme nous l'avons montré à la section 2.4.2, de nombreux facteurs influencent cette bande-passante, ce qui rend cette estimation plus complexe que ce que l'on pourrait penser de prime abord.

De manière similaire, des communications entre le processeur et le coprocesseur sont nécessaires pour initialiser ce dernier et pour informer le premier lorsque le traitement est terminé. La latence de ces communications peut aussi constituer une portion considérable du temps d'exécution de ce coprocesseur si celui-ci est faible. Il faut donc aussi mesurer cette latence.

Nous présenterons ici une méthodologie basée sur l'acquisition automatisée de données expérimentales permettant de caractériser cette bande-passante vers la mémoire ainsi que la latence entre le processeur et le coprocesseur. Nous utiliserons également cette méthodologie pour bâtir un modèle expérimental de performance des communications du Zynq-7020. Ce modèle sera par la suite utilisé au chapitre 5 pour estimer la performance d'accélérateurs sur FPGA par analyse statique.

Rappelons que cette méthodologie suppose un coprocesseur accédant de manière autonome à la mémoire et possédant quelques registres de configuration (par exemple, pour les adresses des données à lire et à écrire) adressables par le processeur. Ce mode de communication correspond à la figure 2.1d du chapitre précédent. Dans le cas du Zynq-7020, cela revient à utiliser les ports HP ou ACP (représentés à la figure 2.6) pour accéder à la mémoire et utiliser un port GP pour permettre au coprocesseur de communiquer avec l'accélérateur.

3.1 Facteurs influençant la caractérisation

Les facteurs, identifiés à la section 2.4.2, influençant la bande-passante vers une mémoire externe atteignable par les coprocesseurs sur FPGA des modèles Zynq-7000 de Xilinx et Cyclone V Sx d'Intel FPGA incluent :

1. La fréquence d'horloge du coprocesseur.
2. Le ou les types de ports de communication fixes choisis (par ex. ACP vs HP sur le Zynq-7000), si le contrôleur mémoire du SoC est utilisé.

3. La largeur (en bits) du port de communication choisi, s'il est configurable.
4. Le contrôleur mémoire utilisé, si ce n'est pas celui du SoC.
5. La quantité de données transférée.
6. Le direction (lecture, écriture ou les deux) du transfert.
7. L'utilisation de transactions en mode rafale ou non.
8. La taille de la cache, s'il s'agit d'un chemin des données cohérent en cache.

Nous croyons que ces facteurs sont présents dans tous les modèles de FPGA ou FPGA-SoC différents. Toutefois, des facteurs additionnels pourraient s'ajouter à des modèles de FPGA-SoC plus complexes. Par exemple, nous savons que l'ajout d'un MMU système sur le chemin des données aura un impact négatif sur les performances [30], bien qu'il puisse simplifier le transfert de ces données ou augmenter la sécurité du système. Cependant, l'étude de ce mécanisme dépasse le cadre de ce mémoire.

3.2 Méthodologie employée pour la caractérisation

Pour pouvoir quantifier l'impact sur la performance de tous les facteurs énoncés à la section 3.1, nous utilisons une approche d'analyse dynamique basée sur l'acquisition de données expérimentales. Nous avons d'abord développé un modèle de coprocesseur paramétrable en fonction des différents facteurs. Nous mesurons ensuite la latence d'exécution du coprocesseur en ne générant que des transactions mémoire et en faisant varier d'un test à l'autre les différents facteurs.

3.2.1 Caractérisation de la latence des communications entre le coprocesseur et sa mémoire

Plus précisément, pour chaque chemin des données possible dans le FPGA ou FPGA-SoC, pour chaque fréquence considérée et pour chaque direction (lecture, écriture, ou lecture et écriture), on génere et instrumente une suite de coprocesseurs effectuant des communications vers la mémoire. Celles-ci sont de taille totale égale à une puissance de 2 ou à 3 $\times$ une puissance de 2, de 1 à M (autrement dit, des tailles de [1, 2, 3, 4, 6, 8, 12, 16, .., M]). Cette suite est répétée pour des communications en une seule rafale, sans rafale et avec des rafales de tailles égales à des puissances de 2 inférieures ou égales à N (soit [2, 4, 8, 16, .., N]).

Une pré-caractérisation doit être effectuée pour déterminer de bonnes valeurs de M et N. Pour la valeur de M, on s'attend à ce que la bande-passante atteignable (c'est-à-dire la latence mesurée divisée par la taille des données transférées) atteigne éventuellement un

plateau. Il suffit donc d'assigner à M une valeur un peu plus grande que la taille du transfert au début de ce plateau. Si le chemin des données implique une cohérence de cache, il faut aussi s'assurer que ce plateau dépasse les dimensions de cette cache. Quant à la valeur N, nous suggérons de tester différentes tailles jusqu'à obtenir une taille de rafale faisant en sorte que la performance du transfert est moins de 10% inférieure à la performance du transfert complet en une seule rafale. Ce dernier nombre a été arbitrairement choisi car il nous semblait être un bon compromis entre la précision de l'estimation et la quantité de tests à effectuer.

Finalement, les mesures devraient être prises plusieurs fois et leur moyenne utilisée pour s'assurer que les données soient en cache (si les transactions sont faites sur un port cohérent avec celle-ci) et limiter l'impact que peuvent avoir les autres composantes du FPGA ou FPGA-SoC utilisant le sous-système mémoire.

Optimisations pour accélérer la caractérisation

Bien que le temps d'exécution de chaque cas de test soit très court, nous devons évaluer l'impact de la direction du transfert, de la fréquence du coprocesseur, du port utilisé, de la taille des données et de la grandeur des rafales, ce qui représente 5 facteurs différents influençant la performance. Une attention particulière doit donc être prise pour réduire l'effort nécessaire et le temps de synthèse de chaque cas de test afin de retarder l'arrivée d'une explosion combinatoire.

D'abord, le processus de génération, d'exécution des tests et de récolte des résultats doit être entièrement automatisé. De plus, si plusieurs plateformes de test (ex. cartes avec le FPGA) identiques sont disponibles, il est recommandé de paralléliser le processus.

Ensuite, il est possible de ne synthétiser qu'un seul coprocesseur pour la majorité des tailles de transfert d'une même fréquence, port, direction et configuration de rafale. Celui-ci ajusterait dynamiquement la taille du transfert en fonction de la valeur d'un registre contrôlé par le processeur. Cette écriture supplémentaire dans ce registre augmente cependant légèrement la latence des communications entre le processeur et le coprocesseur. Il faut donc tout de même synthétiser un coprocesseur par taille de transfert pour les plus petites tailles, c'est-à-dire lorsque l'écriture dans le registre est non négligeable par rapport au transfert total.

3.2.2 Caractérisation de la latence des communications entre le processeur et son coprocesseur

En plus de la latence des communications avec la mémoire obtenue par la méthode décrite précédemment, il faut également mesurer la latence des interactions entre le processeur et le

coprocesseur. Celles-ci incluent l'envoi par le processeur des adresses à accéder en mémoire et une notification par le coprocesseur lorsqu'il a terminé son traitement. Il est particulièrement important de mesurer cette latence lorsque le processeur et le coprocesseur sont faiblement couplés, ce qui est le cas des FPGA-SoC modernes.

Pour isoler la latence des communications entre le processeur et le coprocesseur, il suffit de synthétiser un coprocesseur n'effectuant que ces communications et rien d'autre. Ce coprocesseur recevrait donc une donnée du processeur par cycle puis indiquerait que tout est terminé le cycle suivant. Le processeur pourrait alors appeler ce coprocesseur et mesurer le temps écoulé pendant l'appel. Ce temps variera en fonction de la fréquence du coprocesseur et de la quantité de données d'initialisation envoyées par le processeur (c'est-à-dire les paramètres d'entrée du coprocesseur, comme des scalaires ou des pointeurs vers les données qu'il doit accéder). Le test devrait donc être répété pour toutes les combinaisons possibles, dans la mesure du raisonnable (par exemple jusqu'à 5-6 paramètres).

3.3 Acquisition des données de bande-passante du Zynq-7020

Nous avons appliqué la méthodologie présentée à la section précédente à un Zynq-7020 sur la planchette de développement *Zedboard*. Ceci a principalement été fait pour tester notre méthodologie de caractérisation de la latence des communications d'un coprocesseur avec la mémoire mais cela permet aussi de tester et de mettre en évidence d'éventuelles failles de la méthodologie proposée.

3.3.1 Étude de cas de la méthode de caractérisation sur le Zynq-7000

Les coprocesseurs de test ont été décrits en C/C++ pour synthèse de haut niveau. Ils ont été synthétisés à partir de la plateforme SDSoC 2017.4. Cet outil facilite énormément la génération d'un système complet incluant le coprocesseur synthétisé, l'exécutable le contrôlant et un système Linux complet, le tout dans un format pouvant être copié sur une carte SD puis exécuté sur le *Zedboard*. SDSoC s'occupait aussi de configurer les chemins des données du coprocesseur de manière à ce que ses paramètres de type pointeur communiquent avec la mémoire principale par l'ACP ou les ports HP. Quant aux paramètres scalaires, ils avaient la forme de registres adressables par le coprocesseur via les ports GP du Zynq.

Sélection du chemin des données à tester

L'API de SDSoC offre les fonctions *sds_alloc*() et *sds_alloc_non_cacheable*() pour allouer des blocs de mémoire physiquement contigus (et non contigus par page seulement comme la

fonction *malloc*() sur un hôte Linux). Lors de la génération de la plateforme logicielle/matérielle, SDSoC analyse la provenance des pointeurs passés comme paramètre au coprocesseur généré. Si le pointeur a été alloué par *sds_alloc*(), le chemin des données utilisé pour ce paramètre sera l'ACP. Par contre, si ce pointeur a été alloué par *sds_alloc_non_cacheable*(), ce sera un port HP. De plus, si le coprocesseur a plusieurs paramètres pointeurs connectés par des ports HP, SDSoC séparera ces paramètres de manière à ce qu'ils utilisent chacun un port différent (jusqu'à concurrence des 4 ports disponibles). La sélection du chemin des donnés et de la quantité de ports utilisés par les tests a donc été fait en faisant varier la fonction d'allocation mémoire du logiciel du processeur et le nombre de paramètres du coprocesseur généré.

Détails des coprocesseurs de test

Les coprocesseurs de test sont de simples fonctions C/C++ synthétisés par Vivado HLS via SDSoC. Des exemples de cas de tests en lecture, écriture et lecture-écriture simultanés sont présentés aux figures 3.1a à 3.1f. Notons qu'une macro *define N taille* est implicite aux figures 3.1c à 3.1f.

Les paramètres en entrée et en sortie de chaque coprocesseur sont des entiers de 64 bits afin d'utiliser toute la largeur des ports ACP et HP. Les figures 3.1 montrent aussi que les coprocesseurs de test ne font que des communications mémoire en boucle. La directive de pipelinage qui leur est appliquée fait en sorte que chaque coprocesseur résultant supporte une transaction mémoire par paramètre par cycle. Outre ce cycle par donnée/paramètre, l'exécution de chaque coprocesseur ne nécessite qu'entre 3 et 13 cycles supplémentaires pour l'initialisation, principalement pour remplir le pipeline. La latence des communications de ce coprocesseur est évaluée en soustrayant ce temps d'initialisation au temps d'exécution du coprocesseur.

Un maître AXI est généré pour chaque paramètre de type pointeur. Celui-ci est configuré automatiquement par Vivado HLS pour faire des transactions en rafale si les accès mémoire réalisés sont croissants et contigus. Ceci est le cas de toutes les figures 3.1 à l'exception de 3.1d, où des rafales de taille 8 sont faites car les accès ne sont pas croissants et contigus d'une itération de la boucle principale à l'autre.

De plus, la figure 3.1c montre des communications en lecture/écriture séparées sur deux ports différents. Pour modéliser ceci, nous séparons le transfert en deux tableaux de données deux fois plus petits. Ceux-ci sont accédés en parallèle, puisque la boucle est pipelinée.

Finalement, le seul désavantage notable à utiliser des coprocesseurs décrits en synthèse de

```
#define N 64
void sequentialRW(uint64_t a[N],
                  uint64_t b[N])
{
   for (int i = 0; i < N; ++i)
      #pragma HLS pipeline
      b[i] = a[i];
}
```

(a) Communication en lecture-écriture de taille fixe de 64 en une seule rafale.

```
void sequentialRW(uint64_t a[N],
                  uint64_t b[N],
                  int N)
{
   for (int i = 0; i < N; ++i)
      #pragma HLS pipeline
      b[i] = a[i];
}
```

(b) Communication en lecture-écriture de taille variable, passée en paramètre. La transaction est faite en une seule rafale.

```
void sequentialRW(uint64_t a0[N/2],
                  uint64_t a1[N/2],
                  uint64_t b0[N/2],
                  uint64_t b1[N/2])
{
   for (int i = 0; i < N/2; ++i) {
      #pragma HLS pipeline
      b0[i] = a0[i];
      b1[i] = a1[i];
   }
}
```

(c) Communication en lecture-écriture sur 2 ports en simultané, avec une taille fixe totale N, en une seule rafale.

```
void partialRW_8(uint64_t a[N],
                 uint64_t b[N])
{
   for(int i = N-1; i >= 0; i -= 8) {
      for (int j = 0; j < 8; ++j) {
         #pragma HLS pipeline
         b[i+j] = a[i+j];
      }
   }
}
```

(d) Communication en lecture-écriture, avec une taille fixe totale N, en rafales de 8 éléments.

```
int sequentialR(uint64_t a[N])
{
   int g = 0;
   for (int i = 0; i < N; ++i)
      #pragma HLS pipeline
      g = a[i];
   return g;
}
```

(e) Communication en lecture de taille fixe N en une seule rafale.

```
void sequentialW(uint64_t a[N])
{
   for (int i = 0; i < N; ++i)
      #pragma HLS pipeline
      a[i] = 0xDEADBEEF;
}
```

(f) Communication en écriture de taille fixe N en une seule rafale.

Figure 3.1 Différents coprocesseurs de tests pour évaluer la latence des communications.

haut niveau par rapport à un IP en HDL est visible à la figure 3.1e. Ce coprocesseur ne faisant que des lectures est en effet forcé de retourner une valeur dépendant de ces lectures, ce qui implique un transfert coprocesseur → processeur supplémentaire. Si ce n'est pas fait, Vivado HLS considérera ces accès en lecture seulement comme du code mort et les retirera.

Mesure de la latence d'exécution

Un logiciel exécuté sur le processeur général se charge d'exécuter et de mesurer la latence du coprocesseur. SDSoC permet d'accomplir cette tâche avec une simplicité remarquable : l'exécution du coprocesseur se fait comme l'appel d'une fonction logicielle régulière. Le "compilateur système" se charge ensuite de générer à partir de cet appel le code nécessaire pour exécuter le coprocesseur avec les bons paramètres. Le temps d'exécution de la fonction matérielle est mesuré à l'aide du compteur d'instructions du processeur. Celui-ci est accédé via la fonction *sds_clock_counter*() de l'API de SDSoC.

Quantité de cas de tests

Des tests préliminaires ont montré que la bande-passante atteignable sur le Zynq atteint un plateau aux alentours de transferts de 1 à 2 Mo. Par précaution, nous avons effectué des tests jusqu'à une taille de 8 Mo, c'est-à-dire 1M de transferts de 64 bits. Ceci correspond à la valeur de la variable M à la section 3.2.1. De même, nous avons déterminé que la différence de performance entre un coprocesseur recevant dynamiquement du processeur la taille du transfert à effectuer et un processeur faisant un transfert d'une taille statique était de moins de 1% à partir de transferts de 16 ko (2048 transferts de 8 octets).

Nous avons aussi déterminé que la performance de transactions en rafales de 512 données de 8 octets était à moins de 10% de la performance de transactions en rafales de la taille complète du transfert. Ceci correspond à la valeur N de la section 3.2.1.

Nous avons donc synthétisé des coprocesseurs faisant des accès mémoire de taille statiques de [1, 2, 4, 8, 16, 32, 64, 96, 128, 192, 256, 384, 512, 768, 1024, 1536, 2048] × 8 octets plus un coprocesseur faisant des accès mémoire de taille variable. Ceci fut répété pour les 3 directions possibles du transfert, 4 chemins des données différents (ACP ainsi que 1, 2 et 4 ports HP), pour des communications sans rafale, avec rafale complètes et avec rafales partielles de [2, 4, 8, 16, 32, 64, 128, 256, 512] × 8 octets, à des fréquences de 100 et 142,86 MHz.

Si l'on exclut les cas de tests impossibles (par exemple, on ne peut pas séparer le transfert d'une seule donnée sur 2 ports HP), ceci représente 3197 coprocesseurs de test effectuant 7953 tests différents. Pour limiter le non-déterminisme du système d'exploitation utilisé, le

programme exécute le coprocesseur 1024 fois. Ce programme est lui même exécuté 50 fois, en groupes de 10 exécutions séparées par un délai de 5 secondes. La latence moyenne de la deuxième à la dixième exécution est conservée. La latence de la première exécution est ignorée car nos tests montrent que celle-ci est toujours plus élevée que la moyenne, surtout pour de petites tailles de transfert. Ce comportement est dû à la cache du processeur qui est froide lors des premières itérations, ce qui affecte significativement les résultats.

Génération des coprocesseurs

Le code source du logiciel de contrôle et du coprocesseur de chaque cas de test a été généré à l'aide d'un script Python modifiant un modèle de code pré-défini. Ce script Python fut lui-même exécuté en boucle par un script Bash qui faisait varier les paramètres d'entrée du premier afin de générer tous les cas de test. Ce script Bash était aussi chargé de synthétiser tous les tests en exécutant le "compilateur système" de SDSoC.

Deux stratégies nous ont aidé à accélérer la génération des coprocesseurs. Premièrement, le processus de génération et de synthèse a été parallélisé. Comme il n'y a pas de dépendances de données d'une itération à l'autre de la génération, les seules limites au parallélisme étaient les capacités de calcul à notre disposition. Ensuite, tous les appels au compilateur système de SDSoC ont été faits de manière à utiliser une cache d'IP synthétisés commune. Cela évite de resynthétiser les blocs générés par SDSoC pour connecter le coprocesseur au reste du système.

Ces deux stratégies ont permis d'atteindre un débit de génération d'environ un coprocesseur par minute sur un ordinateur personnel.

Exécution des tests

Par défaut, le *Zedboard* est configuré pour programmer le *bitstream* et démarrer le système Linux présent sur sa carte SD lorsque le système est mis sous tension. Le système de fichiers de cette carte SD est ensuite monté en lecture et écriture par Linux. Comme le *Zedboard* a aussi une connexion Ethernet, il est possible d'automatiser l'exécution des tests à partir d'un ordinateur connecté sur le même réseau. Celui-ci n'a qu'à répéter ces étapes en boucle pour tous les tests :

1. se connecter par SSH au Zedboard.
2. exécuter les tests et sauvegarder le résultat.
3. écraser le fichier *BOOT.bin*, contenant le *bitstream*, et le logiciel de test de la carte SD avec les fichiers du prochain test.

4. redémarrer le Zedboard (qui sera reprogrammé avec le nouveau *bitstream*).

En ayant 2 Zedboards à notre disposition, exécuter les 7953 tests nécessita 2 semaines. La majorité de ce temps fut passé à tester des communications de grandes taille sans rafale ou avec des rafales de petite taille.

3.3.2 Sommaire des résultats de caractérisation

Cette section présente quelques résultats de performance obtenus. En effet, étant donné la grande quantité de données obtenues, nous nous conterons ici de a. donner un sommaire de la situation, b. montrer quelques détails qui ne sont pas disponibles dans la littérature et c. de montrer l'utilité de notre approche par rapport à une estimation ad hoc.

Impact de la taille, fréquence et port du transfert

La figure 3.2 montre la bande-passante bidirectionnelle simultanée obtenue en fonction de la taille du transfert, pour des transferts en une seule rafale. Ces résultats correspondent majoritairement à ce qui était attendu d'après la littérature, c'est-à-dire que :

1. il faut des transferts d'au moins 4 ko pour atteindre une performance raisonnable.
2. passé la moitié de la taille de la cache L2, la performance du port ACP diminue énormément.
3. la performance d'un port HP est majoritairement fonction de la fréquence du coprocesseur.
4. utiliser 4 ports HP sature complètement le contrôleur mémoire : la performance est la même peu importe la fréquence du coprocesseur.

Cependant, nous n'avions pas prévu une aussi faible performance de la configuration à deux ports HP à 142,86 MHz, qui est identique à celle d'un seul port à la même fréquence. Ceci est d'autant plus étonnant que la configuration à quatre ports HP offre une performance supérieure. Nous supposons que ce comportement est dû à l'interconnexion mémoire du Zynq (visible à la figure 2.6 de la revue des concepts), qui combine les ports HP0 et HP1 ainsi que les ports HP2 et HP3 vers deux ports bidirectionnels vers le contrôleur DDR. Or, SDSoC générait un système où les ports HP0 et HP1 (connectés au premier port vers la DDR) n'était utilisés que pour les lectures et les ports HP2 et HP3 (connecté au deuxième port vers la DDR) n'étaient utilisés que pour les écritures. Cette configuration fait de ces ports vers la DDR le goulot d'étranglement de la performance : chacun n'est utilisé que dans une seule direction, ce qui est similaire à n'utiliser qu'un seul port bidirectionnel.

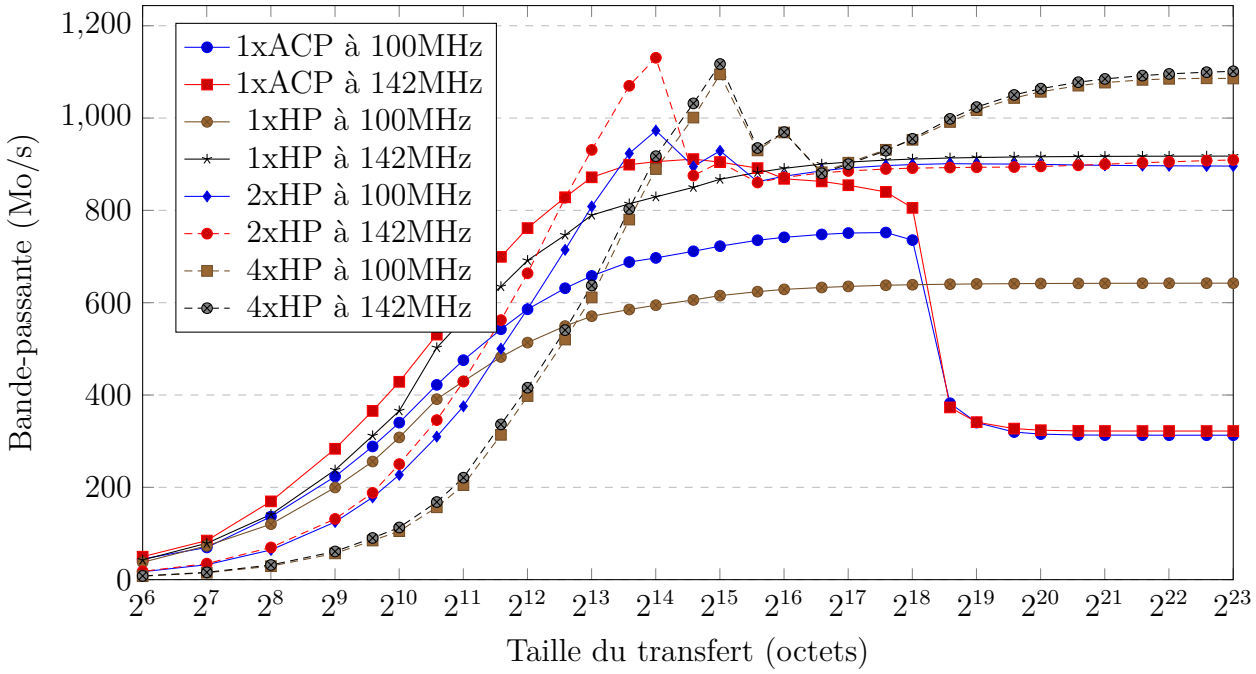


Figure 3.2 Bande-passante bidirectionnelle simultanée en rafale mesurée sur différents ports

Impact des transactions en rafale

Comme le montre la figure 3.3, la taille des rafales effectuées a un impact important sur les performances du transfert. Entre autres, effectuer des transferts sans rafales est 7,5x fois plus lent que d'effectuer des transferts avec. Cependant, les transferts avec rafales de 256 données et plus offrent une performance semblable à des transferts en une seule rafale.

Impact de la direction du transfert

Sur le Zynq-7020, la direction du transfert a aussi un impact sur la bande-passante atteignable : la lecture de données est généralement plus rapide que l'écriture. La figure 3.4a montre que cette affirmation est vraie même pour un seul port HP à 100 MHz, où la lecture est près de 20% plus rapide que l'écriture. Quant à la performance de la lecture-écriture, elle n'est limitée que par cette écriture. Cette figure montre aussi que la lecture sur ce port est limitée uniquement par la fréquence du coprocesseur, atteignant le maximum théorique à cette fréquence de 763 Mo/s.

La figure 3.4b montre qu'à 4 ports utilisés simultanément, les limitations de performance sont dues à d'autres facteurs que la fréquence, puisque la bande-passante en lecture-écriture est de moins de la moitié celle en lecture ou en écriture. Ceci montre aussi que le contrôleur

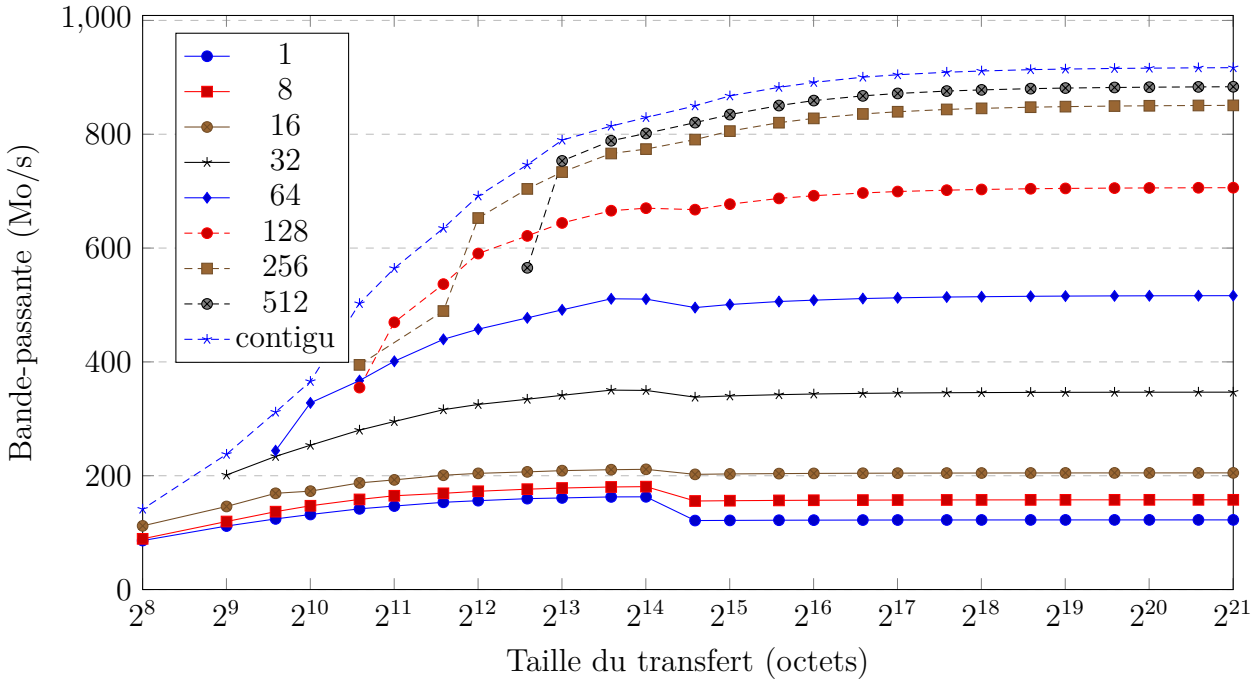
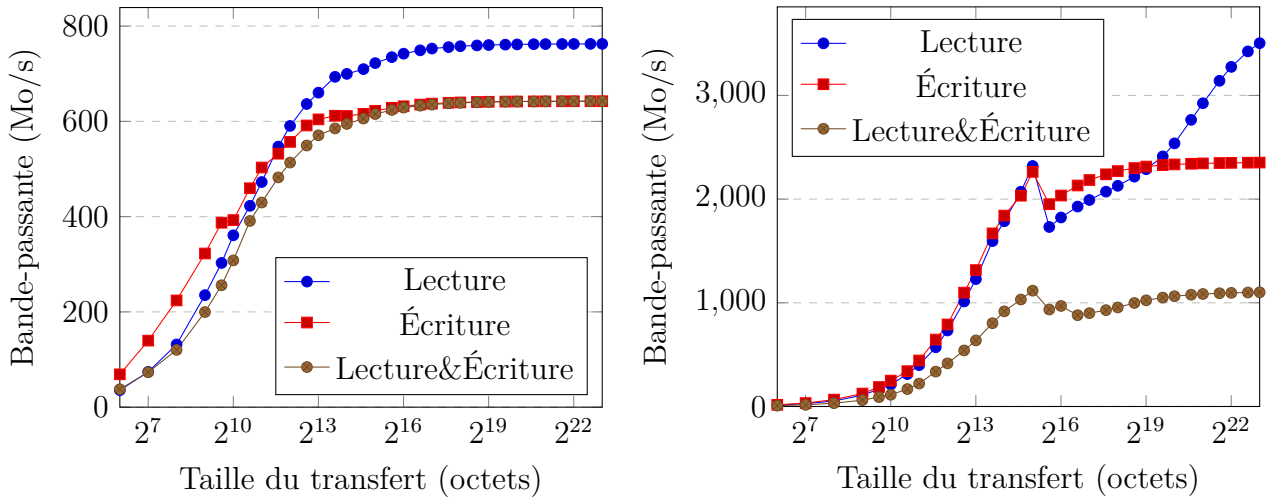


Figure 3.3 BP en lecture-écriture en fonction de la taille (en transferts de 8 octets) des rafales, 1xHP à 142,86 MHz



(a) Pour 1 port HP à 100 MHz

(b) Pour 4 ports HP à 142,86 MHz

Figure 3.4 Bande-passante en lecture, écriture et lecture-écriture

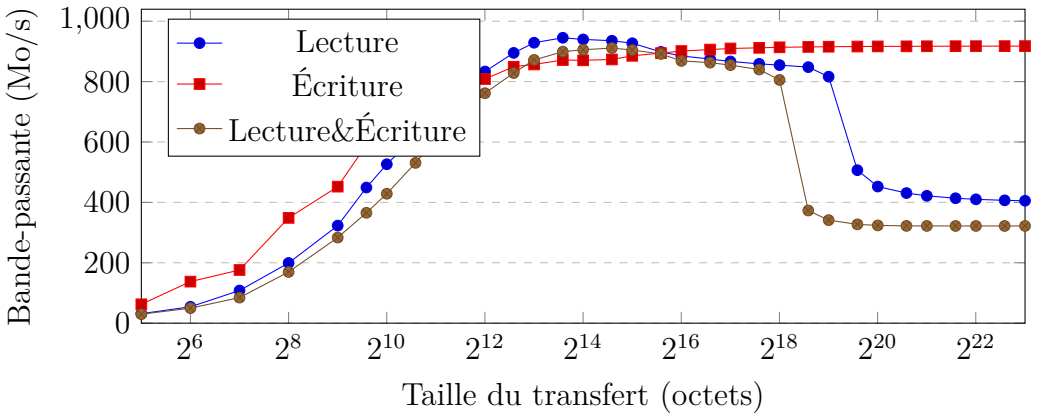


Figure 3.5 Performance de la cohérence de cache à 142,86 MHz

mémoire du SoC ou de la DDR a un comportement sous-idéal lorsqu'il doit arbitrer plus de demandes qu'il ne peut n'en traiter : lire une grande quantité de données d'un coup puis écrire tous les résultats ensuite, serait jusqu'à 40% rapide que de lire les données et d'écrire le résultat simultanément. Cette première option est cependant impossible à mettre en pratique, puisque la portion FPGA est loin d'avoir la quantité de mémoire suffisante pour mettre en cache toutes les données lues et écrites.

Finalement, la figure 3.5 montre que la taille de la cache n'a qu'un effet sur les performances en lecture (et en lecture-écriture). Le fait d'écrire plus de données que la taille de la cache n'a aucun effet mesurable sur les performances d'écriture.

Latence entre le processeur et le coprocesseur

La latence entre le processeur et le coprocesseur dépend de la fréquence du coprocesseur et de la quantité de données à envoyer au coprocesseur. La latence moyenne pour les deux fréquences testées est donnée en cycles du processeur au tableau 3.1. Vivado n'a pas été capable de synthétiser le coprocesseur à 5 arguments à 142,86 MHz généré par SDSoC. Les deux séries de données correspondent à un modèle de régression linéaire avec un $R^2 \geq 0{,}9975$. Ceci confirme que cette latence est directement liée aux nombre de paramètres envoyés au coprocesseur, plus un coût fixe. Celui-ci est probablement dû en grande partie au code d'initialisation et d'attente de résultat du coprocesseur.

Néanmoins, ces données mettent en évidence le couplage faible entre le coprocesseur et le processeur, nécessitant plus de 1000 cycles du processeur par appel, peu importe le nombre d'arguments et la fréquence du coprocesseur. Ceci montre encore une fois qu'un utilisateur cherchant à implémenter des portions d'applications sur un FPGA-SoC doit choisir des

fonctions prenant un temps et une quantité de données considérable s'il veut obtenir une accélération.

Tableau 3.1 Latence moyenne d'appel (en cycles du processeur) d'un coprocesseur sur Zynq-7020 en fonction de la fréquence et du nombre d'arguments.

Fréquence / **Nb. de paramètres**	**100 MHz**	**142,86 MHz**
1	1293	1160
2	1810	1635
3	2251	1999
4	2728	2420
5	3194	*N/A*

CHAPITRE 4 ANALYSE AUTOMATISÉE DES COMMUNICATIONS D'UN COPROCESSEUR

Tel que montré au chapitre 3 et à la sous-section 2.4.2 de la revue des concepts, plusieurs des facteurs qui influent sur la bande-passante atteignable par un coprocesseur sur FPGA dépendent de l'algorithme synthétisé. Il est donc nécessaire d'analyser les communications effectuées par ce dernier. Comme le montrait la figure 1.3, la combinaison de cette analyse et des résultats du chapitre 3 nous permettra d'analyser la latence des communications de coprocesseurs HLS au chapitre 5.

Ce chapitre traite de l'automatisation d'une telle analyse. Après une introduction présentant les requis et approches envisageables pour l'effectuer, nous détaillerons un outil, développé dans le cadre de cette maîtrise, qui réalise cette automatisation par analyse statique du code source de l'algorithme. La description de l'outil inclut aussi des explications détaillées des fonctionnalités des librairies LLVM utilisées pour ce projet.

4.1 Présentation du problème

Nous présentons ici les éléments nécessaires à l'estimation de la bande-passante qui devraient être obtenus d'une analyse des communications d'un algorithme donné. Puis, nous présentons les approches existantes et l'approche utilisée dans ce travail qui permettent de faire cette analyse.

4.1.1 Requis

Pour chaque accès mémoire effectué par l'algorithme, il faut être en mesure d'en :

1. Mesurer la taille (en octets).
2. Compter les répétitions, si l'accès est dans une boucle.
3. Déterminer si les adresses des accès répétés sont croissants et contigus.
4. Identifier la nature (lecture, écriture ou lecture et écriture simultanée).
5. Déterminer la position dans le code.
6. Déterminer si ces transactions sont garanties d'être exécutées ou sont dans un bloc conditionnel.

De plus, l'analyse des accès mémoire doit être suffisamment rapide pour s'insérer dans un processus de conception itératif.

4.1.2 Approches existantes

Plusieurs approches existent déjà pour combler ces requis en tout ou en partie. La plus simple est de synthétiser puis d'implémenter tout le système : la performance de celui-ci est alors disponible directement. Cependant, cette approche a pour désavantage un temps d'exécution significatif, ce qui nuit au processus de conception itératif.

Il est possible de diminuer ce temps d'exécution en procédant plutôt par une simulation du système à un niveau plus abstrait, par exemple à l'aide d'une modélisation au niveau des transactions (TLM) [70] en SystemC. Cette approche peut être réalisée en SystemC directement ou à l'aide d'une plateforme de modélisation d'un système complet comme SpaceStudio [46]. Bien que cette approche puisse simuler rapidement un système complet, son temps d'exécution augmente en $O(\mathrm{N})$ avec le nombre de transactions simulées, ce qui est désavantageux dans les cas où des transactions se retrouvent dans des boucles comportant un grand nombre d'itérations.

Une autre approche, utilisée notamment par l'environnement de développement SDSoC, est d'estimer la bande-passante en ne connaissant que les ports de communication utilisés, la taille totale des tableaux accédés et le sens des transferts. Cependant, cette approche ne prend pas en compte les caractéristiques du transfert qui dépendent du coprocesseur, comme la présence ou non de transfert en rafales et le nombre de fois qu'une transaction vers la même adresse est faite. Cette approche présente donc des lacunes significatives dès que les transactions ne sont pas accédés qu'une seule fois en ordre et de manière contiguë.

Finalement, notons qu'il serait possible de combler tous les requis énoncés précédemment en demandant à l'utilisateur de lire le code source et d'indiquer les accès mémoire qui y sont effectués. Par contre, cette approche devient rapidement laborieuse et propice aux erreurs pour des cas non triviaux.

4.1.3 Approche proposée

Nous proposons une approche par analyse statique automatisée du code pour déterminer les caractéristiques des transactions mémoire effectuées. Cette approche prendrait en entrée le même code source que l'outil de synthèse de haut niveau.

Cette approche a pour avantage principal sa rapidité d'exécution, ne nécessitant pas de synthèse ni de simulation du code source. De plus, son temps d'exécution ne varie pas en fonction du nombre d'itérations de boucles dans lesquelles se trouvent des accès mémoire. Par contre, elle est plus limitée qu'une analyse dynamique lorsque les accès mémoires effectués dépendent de variables dont la valeur n'est connue qu'à l'exécution. Il n'est alors possible que

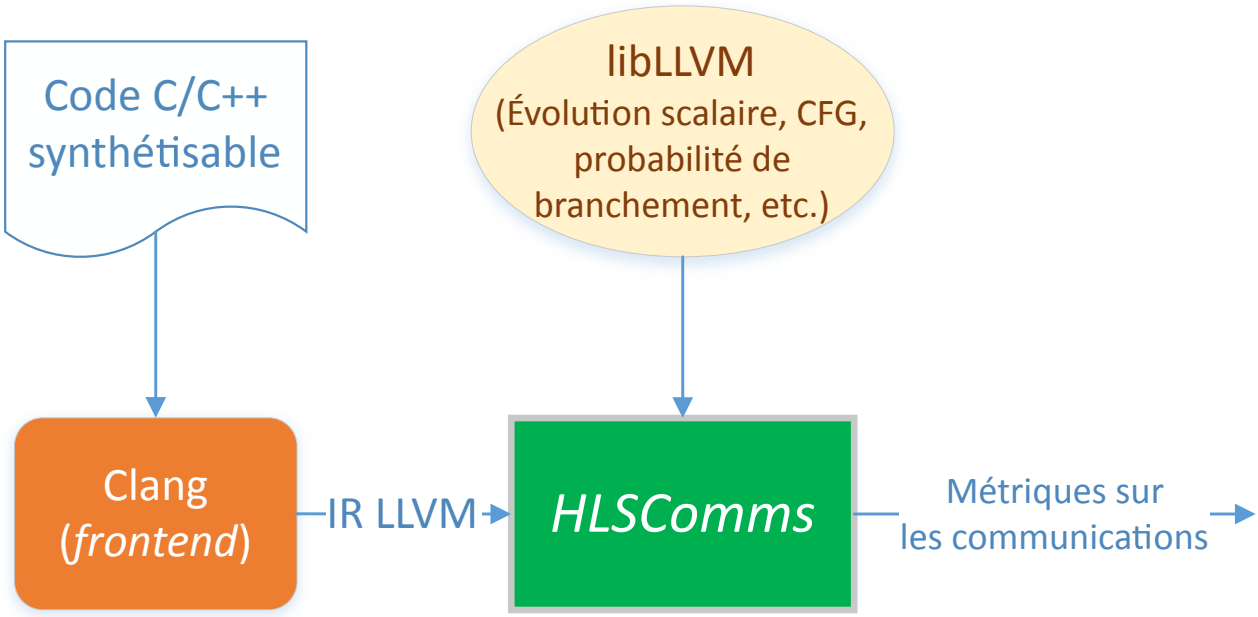


Figure 4.1 Flot d'*HLSComms*, outil d'analyse des communications développé dans le cadre de ce travail

de connaître le pire cas dicté par la taille maximale des variables en question (par ex. 2^{31} -1 pour un entier 32 bits). Cependant, cette limitation n'est pas majeure si l'on considère l'outil développé dans le contexte d'un processus interactif pouvant identifier les variables en cause et permettre une rétroaction par l'utilisateur, qui analyserait manuellement les portions de code concernées et préciserait les valeurs maximales que ces variables peuvent atteindre.

Pour démontrer la faisabilité de cette approche, nous avons développé *HLSComms*, un outil d'analyse statique des communications dont le flot général est présenté à la figure 4.1. Cet outil a été développé en utilisant l'infrastructure de compilateur LLVM. Il est préalablement aidé du transpilateur Clang qui transforme le code source en représentation intermédiaire (IR) d'entrée de LLVM.

Cet outil est présenté en détail à la section 4.2. Des détails supplémentaires sur les fonctionnalités de LLVM utilisées par le projet sont cependant présentées préalablement à la section suivante.

4.1.4 Choix de LLVM par rapport aux alternatives

Le choix de l'utilisation de LLVM, par rapport aux autres possibilités disponibles en logiciel libre (notamment GCC et ROSE), pour réaliser *HLSComms* a été fait en raison de :

1. L'excellence du ratio possibilités offertes / complexité d'utilisation, car

   (a) Le code source est écrit en C++ moderne et facile à comprendre.
   (b) LLVM supporte facilement et efficacement une utilisation comme librairie logicielle.
   (c) Le projet offre des versions stables, incluant une période de tests suffisante, ce qui donne un produit final comportant peu de défauts majeurs.
2. La possibilité de réutilisation d'analyses de code pré-existantes nécessaires aux passes d'optimisation d'un compilateur.
3. La séparation claire entre la partie frontale et finale du compilateur.
4. Son utilisation fréquente en industrie [71], [72].

Les points 1c et 2 sont les principaux avantages favorisant LLVM par rapport à ROSE. Entre autres, les développeurs de ROSE considèrent toute version passant simplement la suite de tests automatisée comme étant stable, ce qui contraste avec et le processus de publication de nouvelles versions de LLVM qui implique de longues périodes de tests impliquant beaucoup plus d'utilisateurs. De même, bien que la documentation de ROSE mentionne la possibilité d'analyse de boucles, les possibilités offertes semblaient plus limitées. En effet, le but principal de ROSE est de permettre des transformations source-à-source de code basées directement sur l'analyse de l'arbre de syntaxe abstrait plutôt qu'une représentation plus simple comme l'IR de LLVM.

Les points 1a, 1b et 3 sont les principaux désavantages GCC par rapport à LLVM. Notamment, le code source de GCC est plus complexe à comprendre pour un novice que LLVM. De plus, bien que GCC offre depuis quelques années une interface sous forme de librairie, libgccjit [73], celle-ci était encore dans un état expérimental au moment d'écrire ces lignes.

De plus, le point 3 permet de baser nos analyses exclusivement sur la représentation intermédiaire du code. Ceci permettrait à *HLSComms* d'être adapté facilement pour analyser du code source écrit dans n'importe quel langage de programmation pour lequel il existe un compilateur frontal générant en sortie une représentation intermédiaire LLVM. Bien que nous n'ayons pas eu le temps d'explorer cette option, ceci permettrait notamment de supporter facilement l'analyse de noyaux OpenCL, qui sont une autre cible fréquente d'outils de synthèse de haut niveau.

Finalement, le point 4 est important car cette utilisation par l'industrie inclut plusieurs outils de synthèse de haut niveau, incluant Vivado HLS ciblé dans ce travail et LegUp. Ceci nous permet donc de faire notre analyse avec un pré-traitement du code semblable à celui fait par l'un de ces outils.

4.2 *HLSComms*

L'outil *HLSComms* détermine par analyse statique les communications effectuées par un algorithme décrit C/C++ destiné à la synthèse de haut niveau. Cet outil prend en entrée le nom du fichier et de la fonction à analyser et produit un fichier JSON contenant les caractéristiques des communications effectuées. Il affiche aussi divers avertissements au besoin, par exemple pour informer l'utilisateur s'il n'est pas capable de déterminer le nombre d'itérations d'une boucle particulière. L'outil est basé sur Clang & LLVM 4.0 et dépend en outre de la librairie *JSON for Modern C++* [74] et des librairies *Filesystem* et *Program Options* de Boost [75]. Une bonne compréhension des fonctionnalités de Clang et de LLVM présentés aux sections 2.6.1 et 2.6.2 est fortement recommandée avant la lecture de cette section du travail.

4.2.1 Vue d'ensemble de l'outil

Le flot général d'*HLSComms* est représenté à la figure 4.2. Avant l'exécution de l'outil, *StructScalarizer*, un préprocesseur optionnel du code source présenté à la section 4.2.7, peut être exécuté pour transformer certaines structures en ensemble de variables indépendantes. *HLSComms* traite ensuite ses arguments en ligne de commande puis utilise Clang comme librairie pour obtenir des informations qui seront perdues dans la transformation en IR LLVM pour finalement compiler la source dans cette représentation intermédiaire.

Les passes d'analyse d'informations de boucles, d'évolution scalaire et de probabilités de branchements ainsi que plusieurs passes d'optimisations fournies par LLVM sont ensuite appliquées à cet IR, suivies de deux passes développées dans le cadre de ce travail : *GEPUniquify* et la passe d'annotations de débogage. *GEPUniquify* transforme l'IR de manière à ce que chaque instruction *GetElementPtr* ne soit utilisée que par une seule instruction subséquente, ce qui est une condition nécessaire à l'analyse des communications. La passe d'annotation de débogage, quant à elle, assigne un nom à tous les blocs de base et redonne à tous les arguments des fonctions leur nom original trouvé dans la phase *FindFunctions*.

La passe *MemoryUse* est ensuite appliquée sur l'IR. C'est cette passe, détaillée à la section 4.2.5 qui fait l'analyse des communications elle-même. Finalement, les caractéristiques de celles-ci sont exportées sous un format JSON.

4.2.2 Phase *FindFunctions*

Certaines informations du code source original se perdent dans la traduction par Clang de ce code en représentation intermédiaire LLVM. Notamment, bien que l'ordre des arguments des

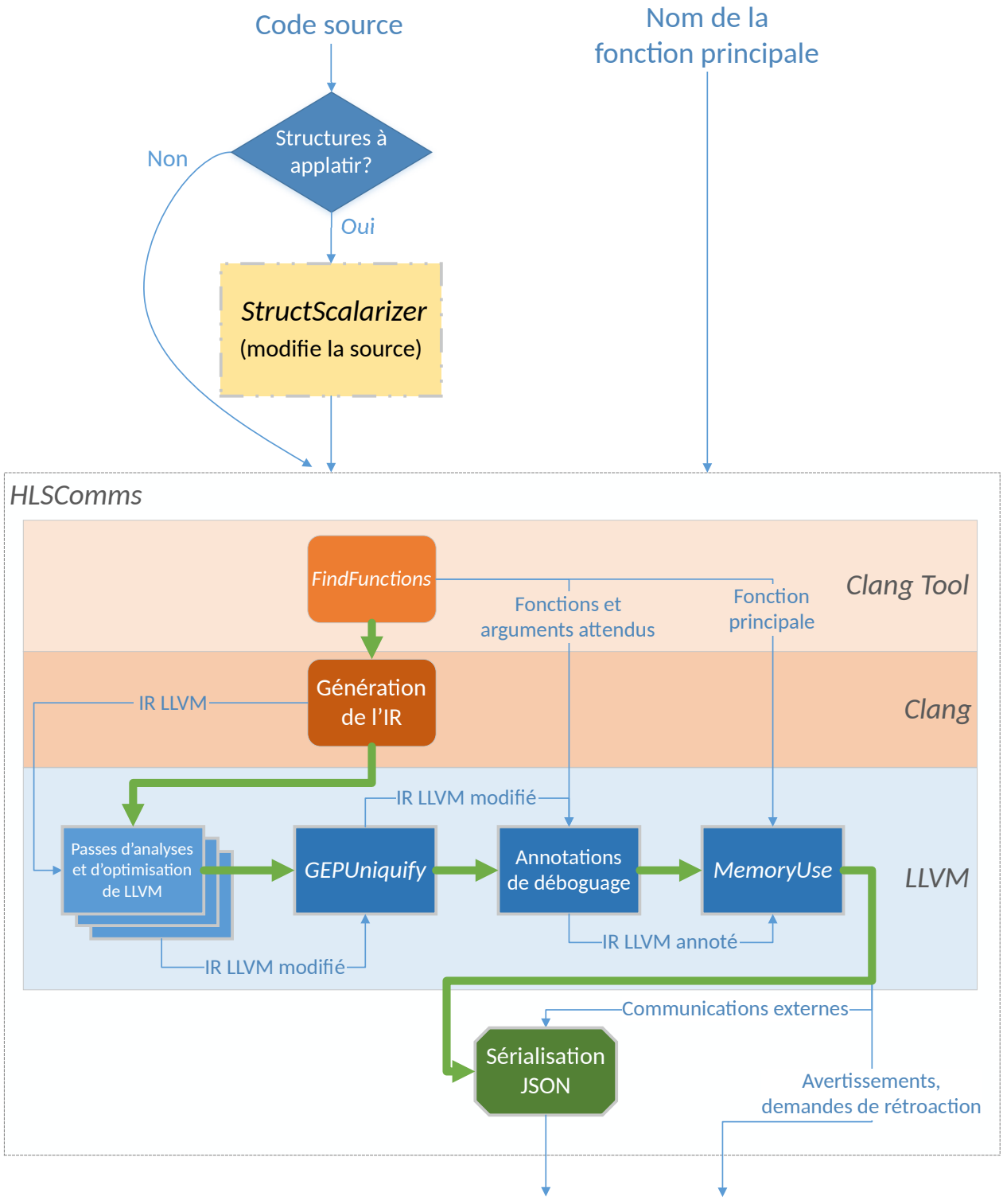


Figure 4.2 Flot interne de l'outil *HLSComms* développé

fonctions soit préservé par la transformation, leur nom ne l'est pas, ce qui est problématique lorsque vient le temps de présenter les résultats à l'utilisateur. De plus, pour simplifier l'utilisation de l'outil, celui-ci prend en entrée le nom de la fonction traitée tel qu'elle est écrite dans le code source, mais ce nom est décoré (*mangled*) dans l'IR si la source est en C++.

Ces informations sont obtenues et les vérifications sont faites par le biais d'un *FrontendAction* qui parcourt l'AST du code source et qui, pour chaque définition de fonction, exécute le *MatchCallback* "FindFunctions". Celui-ci enregistre le nom de la fonction (possiblement décoré) et de ses paramètres dans un conteneur de données réutilisé plus tard. À la fin de l'exécution du *FrontendAction*, nous vérifions que la fonction à analyser se trouve dans ce conteneur et retournons une erreur s'il n'y est pas.

Notons que *FindFunctions* est partiellement basé sur le code source de l'outil de traduction assisté de code C en coprocesseurs développé en [19].

4.2.3 Génération de l'IR LLVM

Pour générer la représentation intermédiaire LLVM à partir du code source fourni, *HLS-Comms* appelle le programme clang-4.0, en lui passant le nom du fichier et les paramètres nécessaires pour qu'il génère l'IR LLVM plutôt qu'un exécutable x86 et qu'il produise un maximum d'informations de débogage.

4.2.4 Passes supportant l'analyse des communications

Trois passes d'analyse et une quinzaine de passes d'optimisation, principalement fournies par LLVM, sont nécessaires pour mener à bien l'analyse effectuée par *MemoryUse* décrite à la section 4.2.5.

Passes d'analyse et d'optimisation fournies par LLVM

Les trois analyses suivantes (présentés à la section 2.6.2) sont nécessaires à l'exécution de *MemoryUse* :

1. L'analyse de l'évolution scalaire des variables.
2. Le calcul des informations de boucles.
3. Le calcul de la probabilité de branchement des différents blocs de base.

Plusieurs passes d'optimisations sont aussi nécessaires à l'obtention de bons résultats. Comme nous prenons en entrée du code source destiné à Vivado HLS, qui est à l'interne très proche

d'un convertisseur IR LLVM vers VHDL/Verilog nous avons décider d'utiliser en grande partie les mêmes passes d'optimisation que cet outil HLS. Nous exécutons donc successivement sur l'IR les passes : d'élimination de code mort, de transformation d'accès mémoire en registres, de combinaison d'instructions redondantes, de simplification des variables d'induction, d'élimination d'instructions *Store* inutiles, de déplacement de code invariant à l'extérieur des boucles, de simplification du flot de contrôle, de propagation de constantes (à l'intérieur d'une même fonction et entre fonctions), d'élimination d'arguments inutiles, de mise en ligne de fonctions, de simplification d'instructions dans des boucles et de suppression de boucles vides.

Passe GEPUniquify

Tel que mentionné précédemment, les opérations de chargement et d'écriture en mémoire ne sont pas modifiées directement d'une itération de boucle à l'autre. C'est plutôt l'une des opérandes de l'instruction *GetElementPtr* (GEP) qui calcule l'adresse du chargement/écriture qui est modifiée. *MemoryUse* utilise donc les instructions GEP pour déterminer les caractéristiques des accès mémoire effectués et ne se sert des instructions de chargement ou d'écriture subséquentes que pour déterminer la direction de l'accès.

Or, dans certains cas (par exemple, lors de l'utilisation de l'opérateur "+="), le même GEP peut-être utilisé pour calculer l'adresse de plus d'un chargement ou écriture. Considérer cette possibilité dans *MemoryUse* complexifierait significativement cette passe, c'est pourquoi il a été préféré de garantir que chaque GEP n'a qu'une seule instruction de chargement ou d'écriture associée.

Cette garantie est apportée par la passe *GEPUniquify*, qui duplique simplement les GEP ayant plus d'une instruction associée et modifie les associations de manière à ce que chacune soit unique.

Ajout d'annotations de débogage

Pour aider au débogage de *MemoryUse*, des informations supplémentaires sont ajoutées à l'IR traité. Notamment, un nom est attribué à chaque bloc de base et les noms des arguments des fonctions, perdus dans la transformation de code source vers IR mais conservés par la phase *FindFunctions* de l'outil, sont réassignés.

4.2.5 Passe LLVM principale : *MemoryUse*

La passe *MemoryUse* est la passe implémentant l'analyse des accès mémoire effectués par le code source fourni. Celle-ci est la dernière passe LLVM exécutée, une fois toutes les analyses et transformations préalables complétées. Nous présentons ici une description générale et simplifiée de cette passe, puis détaillerons chacune de ses sous-composantes.

Notons que chacun des algorithmes en pseudo-code expliqués dans cette section représentent une simplification du code de *MemoryUse*. De plus, dans le but de faciliter la compréhension, le nom et les paramètres de certaines des fonctions appelées en pseudo-code peuvent différer de l'API LLVM réel.

Description simplifiée de la passe

MemoryUse est une passe de type de *FunctionPass*, ce qui assure l'appel par le gestionnaire de passes de sa fonction *runOnFunction*() pour chaque fonction de la source analysée. La figure 4.3 montre un résumé simplifié du comportement de la passe. Le principe est comme suit : pour chaque argument de type pointeur de la fonction, on parcourt les utilisations de ces arguments (c'est-à-dire les endroits où l'argument est utilisé comme opérande d'une instruction). On détermine ensuite si cette utilisation sert au calcul d'une adresse servant à un accès mémoire. Dans le cas positif, on regarde également si cet accès est effectué dans une boucle, dans quel cas on utilise les informations de boucles et d'évolution scalaire fournies par LLVM pour trouver le nombre de répétitions de l'accès et s'ils sont contigus ou non.

Certains éléments de *MemoryUse* ne sont pas représentés à la figure 4.3. Notamment, des opérations doivent être effectués une fois toutes les communications trouvées pour déterminer si une lecture et une écriture peuvent être combinés en lecture/écriture simultanée ou si deux accès mémoire dans la même boucle font en sorte qu'il est impossible d'avoir des accès contigus. Ces cas seront détaillés plus loin.

Détermination des utilisations des arguments

L'API des objets *Value* de LLVM offre la liste de chaque utilisation directe d'arguments et d'instructions dans la fonction analysée. Ceci permet de parcourir chaque instruction utilisant un argument de type pointeur donné, permettant par la suite d'analyser ces instructions pour déterminer les accès mémoire effectués sur ces arguments.

Cependant, n'utiliser que cette liste n'est pas suffisant : l'argument pourrait être modifié avant d'être utilisé, par exemple en incrémentant la valeur pointée comme le montre la figure 4.4a.

```
1  foreach function do
2      foreach argument do
3          if argument.getType() == POINTER then
4              instructions = getInstructionsUsingArg(argument)
5              foreach instruction do
6                  if isAMemoryOperation(instruction) then
7                      if inLoop(instruction) then
8                          addCommunication(getCommunicationFromLoopNest(instruction,
                            argument.getType().getPointedType().getSize()))
9                      else
10                         addCommunication(argument.getType().getPointedType().getSize(),
                            getDirection(instruction), getLocation(instruction),
                            getProbability(instruction))
11                     end
12                 else if isAFunctionCallOperation(instruction) then
13                     calledFunc = getCalledFunction(instruction)
14                     tripcount = getRecursiveLoopTripcount(instruction)
15                     addRelevantFunction(function, calledFunction, tripcount)
16             end
17         else
18             addCommunication(argument.getType().getSize(), READ, 100%)
19         end
20     end
21     if function.getReturnType() != void then
22         addCommunication(function.getReturnType().getSize(), WRITE, 100%)
23 end
```

Figure 4.3 Présentation simplifiée de l'obtention des communications par analyse statique

```
void argModification(int* a)
{
    a++;
    for (int i = 0; i < 12; ++i)
        a[i] = 42;
}
```

(a) Exemple de modification de l'argument avant son utilisation.

```
void whichPointer(int* a, int* b,
                  bool condition) {
    int* c = condition ? a : b;
    for (int i = 0; i < 12; ++i)
        c[i] = 42;
}
```

(b) Exemple d'utilisation conditionnelle d'un argument.

Figure 4.4 Utilisations indirectes d'arguments.

Étant donné la forme SSA de l'IR, c'est-à-dire que chaque valeur ne peut être assignée qu'une seule fois, le résultat de cette modification prendrait obligatoirement la forme d'une nouvelle valeur, dont les utilisateurs doivent aussi être analysés. De plus, plus d'une modification peuvent survenir avant un accès mémoire : le processus d'obtention des utilisateurs de ces modifications doit donc être récursif.

Aussi, l'utilisation d'un argument pourrait être conditionnelle. Par exemple, dans le code C présenté à la figure 4.4b, l'écriture en mémoire peut être faite ou bien par l'argument *a* ou bien par l'argument *b*. Il faut donc tenir compte de la probabilité que l'un soit choisi par rapport à l'autre en se servant des informations de probabilités de branchement.

HLSComms fait toutes les vérifications décrites ici pour déterminer la liste des utilisateurs des arguments. Ceci correspond à la fonction *getInstructionsUsingArg()* de l'algorithme 4.3.

Détermination du nombre d'itérations des boucles

Le nombre d'itérations de chaque boucle est calculée pour chaque sorties de celles-ci par la passe d'évolution scalaire. Il est donc accessible sous la forme d'un objet SCEV (présentés à la section 2.6.2) par sortie. Dans le cas où il y a plus d'une sortie à la boucle, nous avons choisi d'avertir l'utilisateur et de considérer le pire cas possible pour l'analyse.

Tel que montré dans la fonction *getSingleLoopTripcount* à la ligne 13 de l'algorithme 4.5, il suffit donc de regarder pour chacune de ces sorties si le nombre d'itérations est une constante ou est variable. S'il est constant, nous prenons cette valeur. S'il est variable, nous avertissons l'utilisateur puis prenons la plus grande valeur possible que cette variable peut prendre. Dans la majorité des cas, cela correspond à la valeur maximale que le compteur de boucle peut prendre (par exemple 2^{31}-1). Cependant, il peut aussi ne pas y avoir de borne possible, par exemple si le compteur de boucle est modifié dans la boucle.

Les figures 4.6a à 4.6d montrent des exemples du résultat de l'analyse du nombre d'itérations de boucle pour une constante, une variable bornée par 2^{31}-1, une variable non bornée et une variable pouvant prendre une quantité précise de valeur possibles. Notons une particularité du dernier cas : plutôt que de retourner la valeur la plus grande, la passe d'évolution scalaire retourne une valeur correspondant à assigner à "1" le bit le plus significatif de cette plus grande valeur ainsi que tous les bits subséquents. C'est pourquoi, dans l'exemple 4.6d, la valeur maximale retournée est de 15 plutôt que 8. Nous supposons que ce comportement facilite la rapidité ou limite la complexité de l'analyse.

```
1  def getSingleLoopTripcount(loop) :
2      if hasTripcountPragmaHint(loop) then
3          printDebug("Overriding tripcount with user-defined value at " +
             getSourceLocation(use))
4          return getTripcountFromPragma(loop)
5
6      loopExits = loop.getExitBasicBlocks()
7      if loopExits.size() == 0 then  return INFINITE_TRIPCOUNT
8      else if loopExits.size() > 1 then printWarning("Loop has more than one exit,
         considering worst case" + getSourceLocation(loop))
9
10     maxTripcount = 0
11     foreach loopExits do
12         tripcountSCEV = getTripcountFromScalarEvolution(loop, loopExit)
13         if tripcountSCEV.getType() == SCEVConstant  then
14             maxTripcount = max(maxTripcount, tripcountSCEV.getValue())
15         else
16             maxPossibleTripcountSCEV = getMaxPossibleTripcount(loop, loopExit)
17             if maxPossibleTripcountSCEV.getType() == SCEVConstant then
18                 maxTripcount = max(maxTripcount, maxPossibleTripcountSCEV)
19             else
20                 maxTripcount = INFINITE_TRIPCOUNT
21             end
22             printWarning("Can't determine loop tripcount statically, assuming worst
                 case of " + maxTripcount, getSourceLocation(loop))
23         end
24     end
25     return maxTripcount
26
27 def getRecursiveLoopTripcount(loop) :
28     if loop.hasParent() then
29         return getSingleLoopTripcount(loop) *
             getRecursiveLoopTripcount(loop.getParent())
30     else
31         return getSingleLoopTripcount(loop)
32     end
```

Figure 4.5 Obtention du nombre d'itérations d'une boucle ou d'un nid de boucles.

```
void constante(int * data) {
    int nb = 42;
    for (j = 0; j < nb; ++j)
        // L'évolution scalaire
        // retournera 42 comme
        // nombre max. d'itérations
}
```

(a) L'analyse d'itérations de boucle retourne la bonne constante dans des cas comme celui-ci.

```
void variable(int nb, int * data) {
    for (j = 0; j < nb; ++j)
        // L'évolution scalaire
        // retournera  2147483647
        // comme nombre max.
        // d'itérations
}
```

(b) L'analyse d'itérations de boucles ne peut que raisonner sur la valeur maximale d'un entier.

```
void inconnu(int nb, int * data) {
    for (j = 0; j < nb; ++j)
        j = foo();
        // Impossible de connaître
        // la valeur maximale du
        // nombre d'itérations
}
```

(c) Il est parfois impossible de trouver une borne supérieure aux itérations.

```
void choix(bool choix, int* data) {
    int nb = choix ? 3 : 8;
    for (j = 0; j < nb; ++j)
        // L'évolution scalaire
        // retournera 15 comme
        // valeur maximale
}
```

(d) Le cas d'une variable ayant un nombre fixe de possibilités est plus spécial...

Figure 4.6 Différents compteurs de boucle et l'évaluation du nombre maximum d'itérations.

```
for (int i = 0; i < 1920; ++i) {
    for (int j = 0; j < 1080; ++j)
        pixel[i*1080 + j] = foo();
}
```

(a) Double boucle avec accès contigus complets : 1 accès en rafale de taille 1920 × 1080 × sizeof(pixel) accès est fait.

```
for (int i = 0; i < 1920; i +=2) {
    for (int j = 0; j < 1080; ++j)
        pixel[i*1080 + j] = foo();
}
```

(b) Double boucle avec accès contigus par rapport à la première boucle : 1920 accès en rafales de 1080 × sizeof(pixel) octets sont faits.

```
for (int i = 0; i < 1920; ++i) {
    for (int j = 0; j < 1080; j += 2)
        pixel[i*1080 + j] = foo();
}
```

(c) Double boucle avec accès non contigus : 1920 × 1080 accès de sizeof(pixel) octets sont faits.

Figure 4.7 Différentes boucles et le résultat des accès mémoire correspondant.

Détermination des accès mémoire dans des boucles

La détermination des accès mémoire effectuées dans une boucle est faite à l'aide de la fonction *getCommunicationFromLoopNest*() utilisée à la ligne 8 de la figure 4.3. Cette fonction utilise les résultats de l'analyse d'évolution scalaire pour déterminer si l'évolution de la valeur de l'instruction GEP calculant l'adresse de l'accès mémoire fait en sorte que celui-ci est contigu et croissant pour toutes les itérations de la boucle. En d'autres termes, on cherche à savoir si l'évolution de la valeur du GEP est un **SCEVAddRecExpr** dont l'incrément par rapport à chaque itération de la boucle à l'étude est un **SCEVConstant** de taille égale à la taille du type pointé par le GEP.

Si l'accès est contigu et croissant, on considère qu'il s'agit d'un seul accès d'une taille équivalente à la taille de l'objet accédé $\times$ le nombre d'itérations de la boucle dans laquelle il se trouve. Notons que *getCommunicationFromLoopNest*() est récursive : si l'accès est contigu et que la boucle courante a une boucle parente, la fonction est rappelée avec la boucle parente comme paramètre. Par contre, si l'accès n'est pas contigu, on considère qu'il s'agit de d'un accès mémoire correspondant à la taille de l'objet traité répété pour chaque itération de la boucle et de ses parents. En d'autres termes, ce calcul de répétition correspond à l'appel de la fonction *getRecursiveLoopTripcount*() présentée plus haut. L'absence de contiguïté arrête aussi la récursivité de *getCommunicationFromLoopNest*() puisque les accès ne peuvent être contigus par rapport à une boucle parente s'ils ne le sont pas par rapport à la boucle enfant.

Les figures 4.7a à 4.7c donnent une bonne idée des résultats obtenus sur différents pour un accès contigu dans deux boucles imbriquées, dans une seule de deux boucles et dans aucune des deux boucles.

Gestion des probabilités d'accès

Chaque accès mémoire se trouvant dans le code analysé n'est pas nécessairement garanti d'être effectué : notamment, il pourrait se trouver à l'intérieur d'un bloc délimité par un branchement conditionnel. Il est donc nécessaire d'évaluer la probabilité que ces accès se réalisent.

Pour évaluer cette probabilité, nous avons utilisé les résultats de la passe *BranchProbability* de LLVM. Cependant, celle-ci ne fournit que des estimations de la probabilité qu'un bloc de base passe directement à un autre. Nous avons donc implémenté la classe *RecursiveBranchProbabilityInfo*, qui, en ayant accès aux informations de la passe de probabilité de branchement et au bloc d'entrée de la fonction, calcule la probabilité que chaque bloc de la fonction soit exécuté.

Nous utilisions cette classe pour chaque accès mémoire analysé. De plus, si la probabilité de cette accès n'est pas de 100%, nous avertissons l'utilisateur.

Mise en commun d'accès successifs dans la même boucle

L'approche de *MemoryUse* décrite à l'algorithme 4.3, analyse chaque utilisation de chaque argument indépendamment. Cette approche est limitée lorsque plusieurs accès au même argument dans la même direction sont effectués dans une même boucle. Par exemple, la figure 4.8a montre un cas particulier où l'algorithme 4.3 détecterait deux transactions non contigus dans la boucle, bien que les accès mémoires soient en réalité contigus si combinés ensemble.

Cette problématique est résolue par une étape de post-traitement une fois toutes les communications obtenues pour un argument. Nous déterminons pour chaque communication si d'autres communications dans la même direction sont faites dans le même bloc de base. Si oui, nous regardons si la différence entre leur évolutions scalaires fait en sorte qu'il soit possible de les combiner pour obtenir un seul accès augmentant de manière contigu à chaque itération de boucle. Si c'est le cas, nous effectuons cette combinaison.

Invalidation de contiguïté en cas d'accès multiples dans la même boucle

Analyser les communications d'un argument en itérant sur chacune de ses utilisations indépendamment peut aussi causer la détection erronée de transactions contiguës dans une boucle si une autre transaction a aussi lieu dans cette boucle. En effet, si plus d'une lecture ou plus d'une écriture du même argument sont faits dans la même boucle, comme c'est le cas par exemple à la figure 4.8b, cela peut empêcher la génération de transactions en rafale

```
void doubleAccess(int* a) {
    for (int i = 0; i < 42; i+=2) {
        a[i] = 42;
        a[i+1] = 43;
    }
}
```

(a) Situation où l'on doit combiner ces deux transferts non contigus en un transfert contigu.

```
void doubleAccess(int* a) {
    for (int i = 0; i < 42; i++) {
        a[i] = 42;
        a[i+2] = 44;
    }
}
```

(b) Situation où ces deux transferts ne sont pas réellement contigus.

Figure 4.8 Exemples de limites de l'analyse des communications par l'utilisation des arguments seulement.

pour cette direction. Il faut donc rajouter une étape d'invalidation de contiguïté après avoir analysé toutes les communications et fait la mise en commun d'accès successifs.

Pour chacune des communications d'un argument, nous déterminons donc si une autre communication dans la même direction est effectuée dans la même boucle ou une sous-boucle. Si oui, nous transformons la transaction contiguë en une série de transactions non contiguës. De plus, si la communication à l'étude est aussi contiguë au niveau d'itérations de boucles parentes et qu'une autre communication a lieu dans l'une de celles-ci, nous brisons partiellement la contiguïté au niveau de cette boucle parente.

Mise en commun d'accès de directions opposées

Comme nous l'avons montré à la section 3.3.2, la bande-passante en lecture seulement n'est généralement pas la même qu'en écriture ou qu'en lecture-écriture simultanée. Il est donc nécessaire de combiner les lectures et écritures d'un argument qui se trouvent dans le même bloc de base.

Pour ce faire, nous comparons ensemble toutes les communications effectuées par le même argument. Si deux d'entre eux se trouvent dans le même bloc de base et sont égaux en tout point sauf la direction, le deuxième est supprimé et le premier modifié pour indiquer une communication en lecture et écriture simultanée.

Mise en commun des probabilités d'accès mutuellement exclusifs

Des tests sur des cas réels ont montré quelques cas similaires au code présenté à la figure 4.9, où le même accès mémoire est fait dans deux blocs de base différents dont l'exécution est mutuellement exclusive. Plutôt que de traiter ceux-ci comme deux accès différents et ayant

```
void sameCommDifferentBB(int * a, bool condition) {
    if (condition) {
        for (int i = 0; i < 12; ++i)
            a[i] = 42;
    } else {
        for (int i = 0; i < 12; ++i)
            a[i+2] = 43;
    }
}
```

Figure 4.9 Exemple d'accès mémoire mutuellement exclusifs pouvant être combinées.

chacun 50% de chances de se produire, *HLSComms* les traite comme un seul accès ayant 100% de chances de se produire.

Pour faire cette recombinaison, *HLSComms* vérifie, une fois tous les accès mémoire obtenus, si deux accès identiques (même direction et taille) ont lieu dans deux blocs différents qui ne peuvent s'atteindre (c'est-à-dire qu'il n'existe pas de chemin d'un bloc à l'autre dans le graphe des blocs de base de la fonction). Si c'est le cas, ils sont combinés et leur probabilité d'être effectuée est additionnée.

4.2.6 Sérialisation JSON

Dans le but d'être facilement pris en entrée par d'autres outils, les résultats de l'analyse réalisée par *MemoryUse* sont exportés sous forme de texte suivant le standard JSON.

4.2.7 Prétraitement optionnel : *StructScalarizer*

Comme Vivado HLS, *HLSComms* ne supporte pas le déréférencement de pointeurs contenus dans des structures. Cependant dans certains conditions, le premier "aplatit" automatiquement les structures en scalaires indépendants [76, p. 30], ce qui permet de contourner cette limitation. Pour cibler correctement Vivado HLS, *HLSComms* doit donc offrir une fonctionnalité similaire.

Cette fonctionnalité est implémentée via un programme indépendant, *StructScalarizer*, servant de préprocesseur au fichier de code source fourni à *HLSComms*. Il a été développé à l'aide de l'outil *RefactoringTool* et de l'action *MatchFinder* de Clang dans le but de transformer les structures et classes simples spécifiées et utilisées comme paramètres de fonctions en série de composantes scalaires, puis de réécrire le code source correspondant. Par exemple, utiliser cet outil en spécifiant de transformer le paramètre *a* de la fonction *foo* présentée à la figure 4.10a, donne le résultat présenté à la figure 4.10b.

StructScalarizer procède en parcourant successivement trois fois l'AST correspondant au code fourni afin d'exécuter à chaque fois une fonction de rappel différente permettant respectivement de déterminer :

1. toutes les déclarations de paramètres de fonctions.
   - Si le paramètre correspond à la structure concernée, on le modifie pour avoir à la place un paramètre par élément de celle-ci, nommés suivant la nomenclature (*nom du paramètre*)_(*nom de l'élément de la structure*).

```
typedef struct {
        int x;
        float y;
} A;

void foo(A a, int b)
{
        A copie = a;
        a.x = b;
}
```

(a) Fonction avec structures.

```
typedef struct {
        int x;
        float y;
} A;

void foo(int a_x, float a_y, int b)
{
        A copie = { a_x, a_y };
        a_x = b;
}
```

(b) Fonction avec structures aplaties.

Figure 4.10 Structure avant et après sa transformation en scalaires.

2. tous les accès de ces paramètres dans la fonction.
   - On modifie alors cet accès pour remplacer l'opérateur "." par le caractère "_" afin de suivre la nomenclature établie précédemment.
3. tous les utilisations du paramètre dans des assignations.
   - On remplace l'assignation de la structure par une liste d'initialisation correspondant à tous les paramètres aplatis.

On exécute chacun de ces traitements uniquement si le paramètre trouvé et la fonction dans laquelle il se trouve correspondent aux structures/fonctions qu'il a été demandé à l'outil de traiter via une interface en ligne de commande.

Finalement, pour faciliter l'exécution de *StructScalarizer* comme préprocesseur à *HLSComms*, une petite interface fut écrite en Python afin d'appeler successivement les deux exécutables en séparant correctement leurs arguments en ligne de commande respectifs.

4.2.8 Méthodologie de développement

Nous avons suivi le flot de développement suivant pour développer *HLSComms* et *StructScalarizer* :

1. Développer graduellement les fonctionnalités nécessaires en testant sur des cas simples faits pour l'occasion.

2. Lorsqu'un sous-ensemble jugé suffisant des fonctionnalités nécessaires est complété, commencer à tester sur des cas réels.
3. Régler les problèmes rencontrés avec les cas réels en développant des cas de test les isolant.
4. Continuer de développer et de tester itérativement les fonctionnalités manquantes.
5. Tester graduellement sur un plus grand ensemble de cas réels.

De plus, pendant tout ce processus, il était essentiel d'adopter un style de programmation défensif à l'aide d'assertions pour identifier les cas non traités ou non prévus lorsqu'ils sont rencontrés plutôt que de retourner un mauvais résultat sans identifier le problème. Il est aussi essentiel que les cas de tests développés soient re-testés automatiquement après chaque modification pour identifier les régressions possibles.

Notons que dans le cadre de ce projet, le sous-ensemble de fonctionnalités du point 2 consistait à pouvoir déterminer le nombre d'itérations (ou la variable non connue à la compilation si le nombre exact est impossible à déterminer) d'un accès à la mémoire dans un nid de boucles et la contiguïté de cet accès. De plus, le grand ensemble de cas réels correspondait à des exemples typiques de code source d'applications destinées à être synthétisées :

1. Tous les programmes vectoriels de la suite de bancs de test CHStone [49], modifiés pour prendre en entrée/sortie les données nécessaires à leur exécution.
2. Les deux versions de la convolution présentées à la section 1.3 (figure 1.1).
3. Tous les exemples de designs distribués avec l'outil Vivado HLS et SDSoC™, version 2017.4 qui n'utilisent pas la structure *hls : :stream*, *ap_fixed* ou *ap_int*.

Nous considérerons que la faisabilité de l'approche prototypée avec *HLSComms* est démontrée si, pour tous les cas de tests, l'outil est capable de fournir les métriques requises par la section 4.1.1 sur les communications effectuées ou s'il est capable d'identifier la raison pourquoi il ne peut pas le faire. Dans ce dernier cas, il devrait offrir la possibilité de rétroaction par l'utilisateur (par ex. pour un nombre d'itérations de boucle dépendant d'une variable dont la valeur n'est pas connue à la compilation).

La vérification des résultats sera faite en comparant ceux-ci avec une analyse manuelle du code. Une correspondance parfaite pour tous les programmes sera considérée comme démontrant le succès de l'analyseur statique de communications hors-puce. Une erreur dans n'importe lequel des exemples sera au contraire considéré comme un échec.

4.2.9 Limitations

Le but de ce travail n'étant pas une implémentation commerciale, des choix ont dû être faits quant aux fonctionnalités à développer et celles devant être remises à plus tard. Ceci fait en sorte qu'*HLSComms* est présentement limité par les points suivants :

1. Un seul fichier peut être fourni en entrée et ce fichier doit contenir les définitions de toutes les fonctions appelées par la fonction principale.
2. Les pointeurs membres de classes ou de structures ne sont pas supportés.
   — Le préprocesseur *StructScalarizer* (section 4.2.7) peut être utilisé pour contourner partiellement cette limitation.
3. Les pointeurs vers des pointeurs ne sont pas supportés.
4. Les pointeurs de fonctions (incluant les méthodes virtuelles de classes) ne sont pas supportés.
5. Les fonctions ne peuvent retourner qu'un scalaire ou rien du tout.
6. Les fonctions membres de classes ne sont pas supportées.
   — Ceci inclut la structure *hls : :stream* et les classes *ap_fixed* et *ap_int* fréquemment utilisées par Vivado HLS.
7. Chaque accès à un membre d'une structure suppose un accès à la structure entière, ce qui est erroné.
8. Les tableaux à plus d'une dimension ne sont pas supportés.
9. Les structures d'une taille totale de 8 octets ou moins ne sont pas supportées.
10. Les fonctions traitées doivent toutes avoir un nom distinct car l'utilisation de noms décorés (*mangled*) à l'entrée du programme n'est pas supportée.
11. Les directives sous forme de pragma pour Vivado HLS ne sont pas reconnues.

Parmi ces limitations, notons que les points 2 à 5 sont similaires aux limitations imposées par Vivado HLS. Nous jugions donc peu pertinent de tenter de les contourner, surtout qu'elles complexifient significativement le calcul des accès mémoire effectués. Néanmoins, une approche de prétraitement modifiant la source passée à notre outil est envisageable [68] pour contourner ce problème.

De plus, notre solution ne permet pas la détection et le traitement de pragmas personnalisés, incluant les directives Vivado HLS. Cela aurait été utile entre autres pour utiliser les mêmes directives que celles informant Vivado HLS du nombre maximum possible d'itérations de boucles. Cette limitation est due à l'impossibilité d'ajouter des pragmas supplémentaires à

Clang sans modifier et recompiler le code source de celui-ci. Or, nous jugions que dans le cadre d'un prototype, cela ajoutait une trop grande complexité par rapport aux bénéfices possibles.

4.3 Validation de l'analyse des communications produite par *HLSComms*

Pour valider les résultats de l'analyse faite par *HLSComms*, nous avons comparé pour divers cas de tests les résultats obtenus par celui-ci à ceux obtenus par une analyse manuelle. Ces cas de test incluent la suite de tests simples utilisée pour éviter les régressions lors du développement, les exemples de convolution montrés au chapitre 3, les tests vectoriels de la suite de banc de tests CHStone et certains exemples fournis avec Vivado HLS et SDSoC.

4.3.1 Modification apportées aux cas de test

Des modifications mineures ont été apportées aux cas de tests *adpcm*, *aes*, *blowfish*, *jpeg* et *sha* de CHStone. Ces modifications ont été apportées pour ajouter les communications requises au code source. En effet dans leur version originale, les données de ces tests sont synthétisés sous forme de tableaux accessibles directement par l'accélérateur, plutôt que par un pointeur passé en paramètre à la fonction principale. Nous avons modifié la structure du code de manière à ce que ces données soient transmises par pointeur. Ces modifications sont détaillées à l'annexe A.

4.3.2 Comparaison de l'analyse automatisée à une analyse manuelle

Pour tous les cas de tests, nous comparons une analyse manuelle des communications à l'analyse faite par *HLSComms* sur le code source original des tests ainsi que sur version modifiée de celui-ci qui inclut des annotations de nombre d'itérations maximal de boucles. Dans ce dernier cas, nous n'ajoutons ces annotations qu'aux endroits où *HLSComms* a préalablement indiqué qu'il ne pouvait pas déterminer statiquement le nombre d'itérations de boucles.

Suite interne de tests

Pendant le développement d'*HLSComms*, nous avons bâti une suite de 33 tests provenant en majorité de cas réels rencontrés. Cette suite de tests servait à vérifier les fonctionnalités développées et à s'assurer que l'ajout ou la modification d'une fonctionnalité n'entraîne pas de régressions dans des cas déjà testés et fonctionnels. Les communications de tous ces 33 cas de tests sont bien analysées par *HLSComms*.

Opération de convolution

La figure 1.1 présentait un exemple d'une version "naïve" et d'une version aux communications améliorées d'une simple convolution 2D. Nous avons testé avec succès *HLSComms* sur ces deux versions. Pour la première version, l'outil détecte correctement les 1078 écritures de 1918 pixels contigus (les bordures n'étant pas traitées) ainsi que les 6,2 millions de lectures trois pixels contigus. Pour la deuxième version, *HLSComms* détecte correctement une lecture contiguë de 5760 pixels (correspondant à remplir les 3 premières lignes de la cache) suivie de 1078 lectures de 1920 pixels contigus (la première et dernière ligne n'étant pas traitées). Il détecte aussi une écriture contiguë de 2069760 pixels, qui correspond à 1078 lignes de l'image.

Aucune rétroaction par l'utilisateur ne fut nécessaire, puisque la taille de la largeur et de la hauteur de l'image traitée était connue à la compilation.

CHStone

6 des 7 tests vectoriels de CHStone sont analysés correctement par *HLSComms*. Deux de ces six tests, *adpcm* et *jpeg*, produisent un résultat valide sans nécessiter de rétroaction par l'utilisateur car le nombre maximal d'itérations de toutes leurs boucles est connu à la compilation. Cette rétroaction, qui spécifie le nombre maximal d'itérations de certaines boucles, a cependant été nécessaire pour analyser correctement les tests *aes*, *blowfish*, *gsm* et *sha*. Sans cette rétroaction, l'outil affichait une notification indiquant qu'il ne pouvait pas analyser ces cas correctement.

Seul le code source du test *motion* n'a pas pu être analysé en raison de son utilisation de tableaux 3D comme paramètres de fonctions, ce qui n'est pour l'instant pas supporté par *HLSComms*.

Exemples de Vivado HLS et SDSoC

La majorité des exemples fournis avec Vivado HLS et SDSoC n'ont pas pu être testés, car ils font fortement usage des classes génériques *ap_int*, *ap_fixed*, de nombres complexes ou de la structure *hls : :stream*, qui ne sont pas supportés par l'outil. En effet, le préprocesseur *StructScalarizer* est présentement incapable de gérer les appels à des fonctions membres de structures ou classes, ce qui est nécessaire pour ces exemples.

Seuls trois exemples pouvaient être testés : les exemples de port AXI et de filtre par fenêtrage de Vivado HLS ainsi que l'exemple de multiplication matricielle de SDSoC. Ces tests ont tous obtenus le bon résultat sans modifications, les bornes de boucles étant toujours constantes.

4.4 Résumé

Nous avons présenté dans ce chapitre *HLSComms*, un outil se servant de LLVM et Clang afin d'analyser de manière statique les communications faites par un coprocesseur HLS. Pour chaque communication, *HLSComms* donne des informations sur tous les facteurs dépendant du code pouvant influencer la performance des communications. Il fournit donc la direction de cette communication, sa taille, son nombre de répétitions, sa contiguïté, la probabilité que celle-ci soit effectuée et sa position dans le code.

La composante principale d'*HLSComms* est la passe LLVM *MemoryUse*, qui se sert elle-même du résultat de certaines passes d'analyse fournies avec LLVM. Parmi celles-ci, la plus importante est la passe d'évolution scalaire, qui permet de déterminer le nombre d'accès dans des boucles et d'évaluer l'évolution des adresses accédées en mémoire au fur et à mesure de l'exécution du programme.

Nous avons validé que le résultat produit par*HLSComms* était identique à celui produit par une analyse manuelle du code pour plusieurs cas de tests, qui sont présentés à la section 4.3.2. Dans ces tests, s'il n'était pas possible pour l'outil de déterminer statiquement le nombre d'itérations de certaines boucles, une notification était envoyée à l'utilisateur afin qu'il détermine celui-ci manuellement et en informe l'outil.

CHAPITRE 5 ESTIMATION DE LA LATENCE D'EXÉCUTION DE COPROCESSEURS PAR ANALYSE STATIQUE

Ce chapitre combine les données expérimentales de latence des communications du Zynq-7020 acquises au chapitre 3 et l'analyse automatisée de communications de coprocesseurs présentée au chapitre 4 pour estimer de manière statique la latence d'exécution de coprocesseurs. Nous y présentons la procédure de combinaison, suivie de tests comparant la latence d'exécution estimée et réelle des coprocesseurs utilisés au chapitre précédent pour valider *HLSComms*. Ces tests montrent que pour les coprocesseurs dont la latence d'exécution est limitée par la bande-passante vers la mémoire et pour lesquels les boucles effectuant les communications sont pipelinées, l'erreur relative de l'estimation est en moyenne de 8,5% et varie de 0,2 à 25%. Le temps d'exécution de cette estimation est au minimum un ordre de grandeur plus petit que le temps de synthèse suivi de l'exécution du coprocesseur.

5.1 Intégration des différents délais pour estimer la latence des communications

Pour mesurer la latence d'exécution de coprocesseurs décrits en C/C++ synthétisable, il faut connaître la latence :

1. de la micro-architecture, fournie par l'outil HLS.
2. des communications, obtenue par la mise en commun :
   — des données de caractérisation de la latence des communications du FPGA.
   — des communications faites par le coprocesseur.

Le premier point correspond à l'estimation de latence fournie par l'outil HLS. Celle-ci inclut les délais liés au contrôle des interfaces de communication, mais pas ceux des communications elle-mêmes, puisque l'outil HLS n'a aucune notion du reste du système. Notre approche nécessite aussi d'avoir une estimation de la latence de chaque nid de boucles du coprocesseur individuellement. Plusieurs outils HLS, comme Vivado HLS utilisé ici, supportent cette fonctionnalité.

Le deuxième point consiste en l'obtention de la latence des communications du coprocesseur en faisant correspondre les caractéristiques de ses communications (fournies par *HLSComms*) aux données de caractérisation des communications du FPGA présentée au chapitre 3.

Notons que bien que l'obtention de la latence de la micro-architecture, des caractéristiques des communications et de la caractérisation du FPGA soit automatisée, la combinaison des

trois pour obtenir la latence d'exécution ne l'est présentement pas. Ce choix a été fait pour d'abord prioriser le développement du reste des éléments présentés dans ce mémoire.

5.1.1 Estimation de la latence d'exécution

Nous émettons l'hypothèse que le goulot d'étranglement de la performance de chaque nid de boucle exécuté séquentiellement par le coprocesseur est ou bien dans l'obtention des données et l'écriture des résultats, ou bien dans le traitement de ces données. Cette hypothèse suppose la présence d'un pipeline permettant de lire des données, d'en traiter et d'écrire des résultats simultanément. Bien qu'il soit possible de concevoir des coprocesseurs ne suivant pas ce modèle, nous croyons que cela ne représente qu'une minorité des cas, puisque ce parallélisme est de loin le meilleur moyen d'obtenir une accélération par rapport à une exécution sur un processeur général [77, p.48]. Cette hypothèse suppose aussi que chaque nid de boucle différent est exécuté séquentiellement. Ceci est le comportement par défaut de plusieurs outils HLS, incluant Vivado HLS, car des nids de boucles successifs ont généralement des dépendances de données entre eux. À moins de mention contraire, les tests présentés à la section 5.2 respectent ces deux hypothèses.

Cependant, les coprocesseurs ne passent pas la totalité de leur temps dans des boucles : des communications et du traitement de données peuvent aussi être faits à l'extérieur de celles-ci. Nous approximons ce temps comme étant la somme du temps passé à faire des communications hors boucle et du temps passé à traiter ces données. Ceci peut être inexact : des communications peuvent être effectuées en parallèle à un traitement d'autres données. Toutefois, nous croyons que la différence sera négligeable puisque dans la très grande majorité des cas, le temps passé dans des boucles est très supérieur au temps passé à l'extérieur de boucles.

Nous estimons donc la latence d'exécution totale du coprocesseur t_{tot} à :

$$t_{\text{tot}} = \left(\sum_{i=0}^{N} \max \begin{Bmatrix} t_{\mu\text{arch}}(i) \\ t_{\text{comms}}(i) \end{Bmatrix} \right) + t_{\mu\text{arch scal}} + t_{\text{comms scal}} \tag{5.1}$$

où N est le nombre de nids de boucles, $t_{\mu\text{arch}}(i)$ la latence de la micro-architecture de la boucle i, $t_{\text{comms}}(i)$ la latence des communications de cette même boucle, alors que $t_{\mu\text{arch scal}}$ et $t_{\text{comms scal}}$ représentent la latence de la portion à l'extérieur de boucles de la micro-architecture et des communications.

5.1.2 Obtention de la latence des communications

Les caractéristiques des communications d'un coprocesseur donné correspondent rarement exactement à celles de mesures présente dans les données expérimentales recueillies. Il faut donc faire certaines approximations pour estimer la latence résultante de ces communications à partir des données expérimentales.

D'abord, nous interpolons linéairement la latence de communications dont les caractéristiques se trouvent entre deux points mesurés expérimentalement. Par exemple, la latence d'un transfert en rafales de 3 éléments sera obtenue en faisant la moyenne de la latence d'un transfert en rafales de 2 éléments et d'un transfert en rafales de 4 éléments.

Ensuite, nous multiplions la latence estimée de chaque communication par la probabilité que cette communication soit effectuée. Dans la majorité des cas, cette probabilité est de 100%, mais celle-ci peut-être inférieure si la communication est dans un bloc conditionnel.

De plus, s'il y a contention des ressources et que plusieurs variables partagent le même port dans la même direction, nous considérerons que la latence de ces accès correspond à celle d'un accès d'une taille représentant la somme des tailles des accès réalisés sur ce port. Par exemple, 2 transferts contigus d'une taille de 8192 octets réalisés en parallèle sur le même port seront considérés comme un seul transfert contigu de 16384 octets.

Par contre, s'il n'y a pas de contention de ressources et que plusieurs variables accèdent simultanément à la mémoire dans la même direction sur des ports différents, nous estimons la latence de ces accès comme celle d'un seul accès sur tous les ports utilisés. Par exemple, nous considérons 2 lectures de la même taille dans le même nid de boucle comme équivalent à une lecture de cette taille sur 2 ports différents. Dans le cas où le nombre de lectures ne correspond pas au nombre d'écritures, nous faisons une moyenne du résultat si le nombre d'écritures correspondait au nombre de lectures et si le nombre de lectures correspondant au nombre d'écritures.

Finalement, si une communication est faite en lecture-écriture mais que l'une des deux directions fait des transferts avec des rafales de plus petite taille que l'autre, nous considérons que le transfert n'est fait que dans la direction ayant la plus petite taille de rafale. Le raisonnement derrière cette approximation est que le goulot d'étranglement risque fort de passer par la plus petite rafale et que l'impact d'un transfert avec de plus grosses rafales en direction opposée sera moins important.

5.2 Résultats

Nous présentons ici l'erreur relative de l'estimation de performance par rapport au temps d'exécution réel de coprocesseurs. À moins de mention contraire, toutes les boucles de ces coprocesseurs qui effectuent des communications ont été pipelinées (en ajoutant la directive *HLS pipeline* au code) et chaque paramètre de type pointeur pointe vers des éléments de 64 bits. Tous les coprocesseurs testés ont été synthétisés à l'aide de SDSoC 2017.4.

5.2.1 Résultats préliminaires sur le filtre Sobel naïf

Nous avons d'abord testé la version naïve "logicielle" de la convolution présentée à la figure 1.1 de l'introduction. Celle-ci est un bon exemple de coprocesseur limité en bande-passante puisqu'elle n'a aucune cache d'inférée, ce qui implique de lire en mémoire chaque pixel 9 fois.

Première version

L'analyse des communications effectuée par *HLSComms* montrait que l'entier 8 bits en entrée était lu en 6 202 812 rafales de 3 éléments et que l'entier 8 bits en sortie était écrit en 1078 rafales de 1918 éléments. L'implémentation de l'algorithme ne comporte qu'un seul nid de boucle : la latence estimée correspondra donc essentiellement au maximum entre la latence des communications et la latence de la micro-architecture. De plus, comme la taille des rafales en lecture est petite en comparaison de la taille des rafales en écriture, nous utilisons les données de latence en lecture pour estimer la latence des communications.

Pour un port HP à 142,86 MHz, l'estimation de latence de la micro-architecture est de 116 ms pour le nid de boucles et ≈ 0 pour le reste. Quant à la latence calculée des communications, elle est de 93,8 ms. Cependant, le temps d'exécution réel est de 1,22 secondes, ce qui ne correspond pas du tout à l'estimation.

La raison de cette différence est une mauvaise modélisation des communications : celle-ci avait été faite avec des transactions prenant tous les 64 bits de l'interface alors que les transactions n'ont ici que 8 bits de large. Cependant, même en multipliant par 8 la latence des communications pour tenter d'émuler un transfert de 64 bits, l'estimation ne donnerait que 750 ms, ce qui est significativement inférieur au résultat obtenu.

Ajustement de la largeur des données

Nous avons donc modifié le code source du coprocesseur pour que le type de ses paramètres soit des pointeurs de 64 bits plutôt que 8. Cela change le comportement du coprocesseur,

mais nous ne sommes intéressés ici que par la latence d'exécution. Nous avons ensuite estimé la latence d'exécution pour toutes les combinaisons de ports et de fréquences modélisées au chapitre 3. Les résultats, beaucoup plus concluants, sont présentés dans le tableau 5.1. Notons que dans tous les cas, la latence estimée du nid de boucle est égale à la latence des communications. Il n'y a effectivement pas de communications à l'extérieur de cette boucle et la latence d'initialisation du coprocesseur est négligeable.

À l'exception de la configuration ACP à 142,86 MHz et des configurations à 4 ports HP, le temps estimé est très proche de la réalité. Nous ne sommes pas certains de la raison expliquant l'écart entre la prévision et la réalité pour l'ACP. Il est possible que le fait de relire plusieurs fois les mêmes données ait eu un effet bénéfique sur la disponibilité de ces données en cache. Cependant, c'est ce même port à 100 MHz qui donne l'estimation la plus précise, ce qui met à mal cette hypothèse.

Quant aux quadruples ports HP, nous supposons que cette configuration sature beaucoup plus le système mémoire et donc que de supposer que les écritures en rafales de 1918 éléments sont négligeables par rapport aux lectures en rafales de 3 éléments n'est pas la bonne marche à suivre. En effet, le rapport entre la latence réelle de la configuration 4x HP et 2x HP des deux fréquences (124 et 121%) est très similaire au rapport de la performance de ces ports dans une configuration de lecture & écriture contiguë (121 et 121%). Il est cependant différent de celui de lectures de petites rafales, où la plus petite bande-passante atteinte sature moins le contrôleur mémoire et permet une meilleure performance de la configuration 4x HP relativement à 2x HP (158 et 156%). Pour confirmer cette hypothèse, nous avons synthétisé deux coprocesseurs de test semblables à ceux décrits à la figure 3.1, mais faisant des lectures non contiguës tout en faisant des écritures contiguës. Utiliser ces données à la

Tableau 5.1 Latences estimées et latences réelles pour un filtre de Sobel "naïf"

Port	**Fréquence (MHz)**	**Estimation de la μ-architecture (s)**	**Estimation des communications (s)**	**Latence réelle (s)**	**Erreur (%)**
ACP	100,00	0,165	**0,998**	0,995	0,3
ACP	142,86	0,116	**0,854**	0,729	17,1
1x HP	100,00	0,165	**0,959**	0,966	-0,7
1x HP	142,86	0,116	**0,750**	0,763	-1,6
2x HP	100,00	0,083	**0,560**	0,576	-2,7
2x HP	142,86	0,058	**0,473**	0,480	-1,3
4x HP	100,00	0,041	**0,350**	0,463	-24,5
4x HP	142,86	0,029	**0,300**	0,396	-24,2

place réduit l'erreur à 9% et 15 %, ce est une amélioration mais indique tout de même une perte de précision de la caractérisation quand le contrôleur mémoire est saturé.

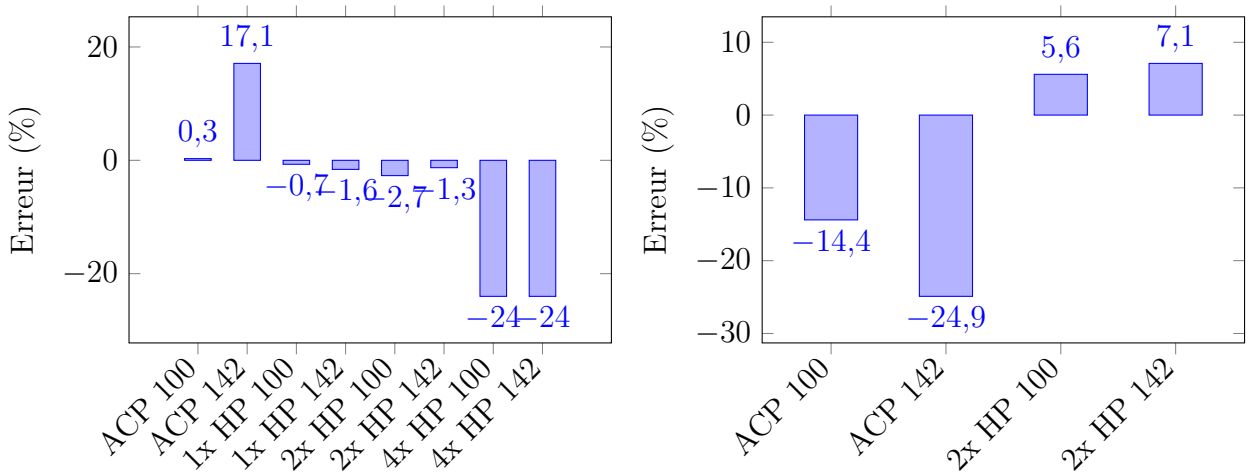
Le faible taux d'erreur de ces résultats par rapport à ceux du coprocesseur 8 bits indique que la performance d'accès mémoire non alignés est significativement inférieure à la performance d'accès mémoire alignés sur 64 bits. On ne peut donc pas estimer la performance de coprocesseurs faisant des transferts de moins de 64 bits avec les données expérimentales du chapitre 3. Les coprocesseurs de tous les autres tests présentés dans ce chapitre ont donc été modifiés pour faire des accès mémoire de 64 bits.

5.2.2 Coprocesseurs limités par la mémoire

Le filtre de Sobel naïf présenté plus haut ainsi que l'exemple d'addition et de multiplication matricielle de SDSoC 2017.4, l'exemple de filtre 1D de Vivado HLS 2017.4 et les tests *GSM* et *AES* de CHStone sont tous limités par la bande passante vers la mémoire. Les tests de multiplication matricielle, *GSM* et *AES* comportent d'ailleurs plusieurs boucles réalisant des communications, mais chacune d'entre elles est limité par la bande-passante vers la mémoire. Les figures 5.1a à 5.1f montrent l'erreur relative de l'estimation de latence par rapport à la réalité pour tous ces exemples. Une erreur positive montre une latence prévue plus élevée que la réalité et vice versa. Ces résultats montrent tous les ports pour lequel un coprocesseur de test a pu être synthétisé : certains ne pouvaient pas être séparés sur plusieurs ports HP ou, dans le cas de la figure 5.1e, synthétisés à 142 MHz.

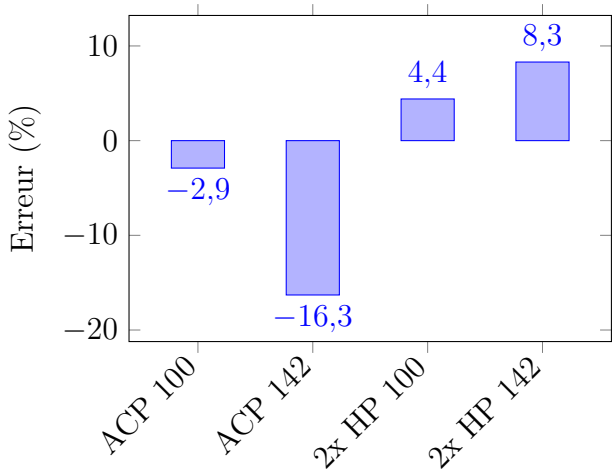
Notons que la performance de l'exemple de maître AXI de Vivado HLS 2017.4 est aussi limitée par les communications vers la mémoire. Cependant, ce test ne fait que copier des données de la mémoire puis écrire un résultat, ce qui très similaire au comportement des coprocesseurs de test utilisés pour caractériser la latence des communications. Nous avons donc exclu ce test de nos résultats pour éviter de les biaiser par l'erreur relative artificiellement basse qu'aurait eu ce test.

Les additions et multiplications matricielles des figures 5.1b et 5.1c ont deux matrices 32x32 en entrée et une matrice 32x32 en sortie. Le coprocesseur de multiplication est séparée en deux boucles, la première copiant les données des deux entrées dans une cache en BRAM et la seconde effectuant la multiplication et écrivant le résultat, le tout dans un pipeline sortant 1 donnée par cycle. Nous considérons donc les configurations ACP comme faisant une lecture de 16384 octets suivie d'une écriture de 8192 octets et les configuration HP comme faisant une lecture de 16384 octets séparée sur 2 ports suivi d'une écriture de 8192 octets sur un seul port. Quant à l'addition matricielle, elle est implémentée en une seule boucle. Nous avons donc pris la moyenne d'une lecture-écriture de 16384 octets séparés sur 2 ports et d'une

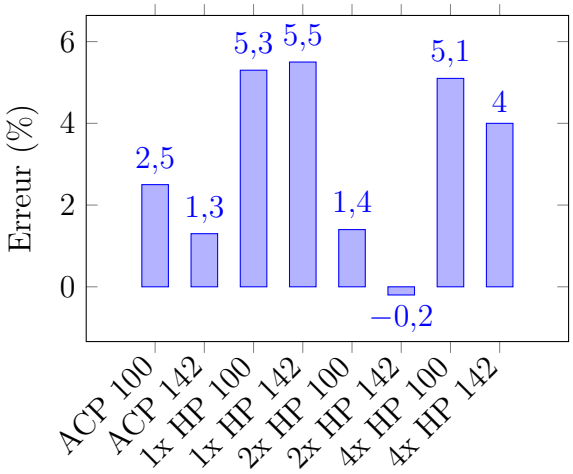


(a) Filtre de Sobel "naïf" modifié

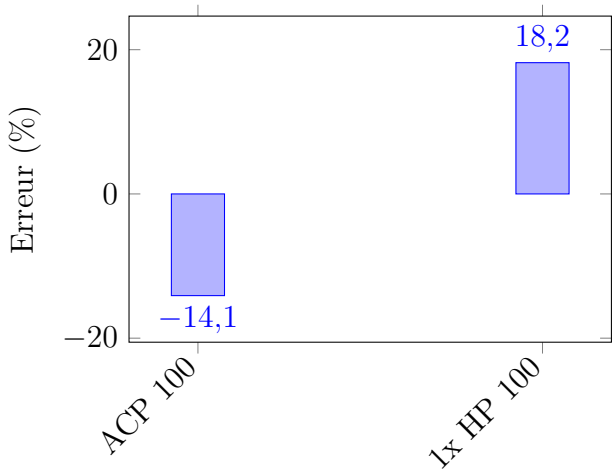
(b) Multiplication matricielle



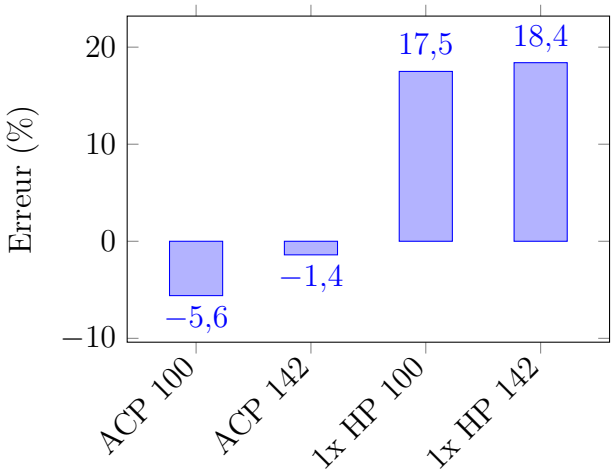
(c) Addition matricielle



(d) FIR 1D



(e) Test GSM de CHStone



(f) Test de cryptage AES de CHStone

Figure 5.1 Erreurs relatives entre l'estimation et la réalité

lecture équivalente séparée sur 2 ports pour la configuration HP, et finalement considéré la somme d'une lecture de 8192 octets et d'une lecture-écriture de 8192 octets pour l'ACP.

La figure 5.1d présente l'erreur d'une simple application d'un filtre par fenêtrage 1D sur 32 octets. Celui-ci est tiré d'un exemple de Vivado HLS et a été modifié pour utiliser des données de 64 bits. Bien que le coprocesseur généré soit assez similaire aux coprocesseurs de tests générés au chapitre 3, ces résultats montrent qu'il est possible d'obtenir des résultats précis malgré la petite taille du transfert.

Finalement, les figures 5.1e et 5.1f illustrent les résultats les plus représentatifs de "cas réels" puisqu'il s'agit d'applications un peu plus complexes qui ne sont pas basées seulement sur une ou deux boucles faisant des communications et des calculs simultanément. Notamment, le code source de chacun de ces deux tests fait plusieurs centaines de lignes de code et une portion significative de la latence de la micro-architecture et des communications est à l'extérieur de boucles. De plus, certaines de ces communications se trouvent dans des blocs conditionnels et ne sont donc pas garanties d'être exécutées. Dans le cas d'*AES*, le nombre d'itérations de certaines boucles est aussi variable en fonction des entrées du coprocesseur.

Compte tenu de ces circonstances, nous croyons que les erreurs obtenues pour ces deux derniers tests sont acceptables et démontrent la faisabilité de l'approche d'estimation des performances par analyse statique. Cela étant dit, l'écart entre les deux résultats du test *GSM* montrent qu'il est encore possible d'améliorer la précision de la caractérisation des communications. En effet, les latences réelles entre l'ACP et le port HP pour ce test sont similaires à 106,6 et 102 μs respectivement, mais les données expérimentales de bande-passante de ces deux ports à ces tailles de transfert prévoyaient une différence de la performance de plus de 30%.

Importance du pipeline

Comme il a été mentionné plus haut, le code source de tous les coprocesseurs testé a été annoté de manière à forcer Vivado HLS à pipeliner toutes les boucles effectuant des communications. Ceci permet au coprocesseur de faire les communications de certaines itérations de boucles en même temps que le traitement des données d'autres itérations. Il n'y a généralement pas de désavantage à appliquer cette directive : elle augmente habituellement les performances sans augmenter la consommation de ressources.

Cela étant dit, si les boucles faisant les communications ne sont pas pipelinés, l'hypothèse que le goulot d'étrangement d'une boucle est ou bien les communications ou bien les calculs effectués tombe et la procédure de combinaison décrite à la section 5.1 ne permet pas d'estimer

la latence d'exécution correctement. Par exemple, utiliser la procédure d'estimation décrite plus haut sur une version sans pipeline du test *GSM* sous-estime la latence d'exécution de 48% pour la configuration ACP et de 30% pour la configuration HP.

5.2.3 Coprocesseurs limités par la capacité de calcul

La performance du reste des tests de CHStone et le filtre de Sobel modifié de la figure 1.1 est limitée par la capacité de calcul des coprocesseurs correspondant. La latence d'exécution résultante est très similaires à l'estimation du nombre cycles d'exécution de Vivado HLS. Cela confirme que l'estimation de la latence de la micro-architecture est très proche de la réalité si le goulot d'étranglement de la performance n'est pas dans les communications.

5.3 Résumé

Nous avons combiné dans ce chapitre les travaux présentés aux chapitres 3 et 4 pour déterminer statiquement en une à deux secondes la latence des communications faites par un coprocesseur décrit en synthèse de haut niveau. Nous avons aussi développé une méthode d'estimation de la latence d'exécution du coprocesseur, qui combine cette latence des communications et la latence de la micro-architecture fournie par l'outil HLS. Nous avons montré qu'il est ainsi possible d'estimer en quelques dizaines de secondes le temps d'exécution de ce coprocesseur. L'erreur moyenne de cette estimation est de 9% et l'erreur maximale de 25%, si la performance de ce coprocesseur est limitée par les communications avec la mémoire, que les boucles faisant ces communications sont pipelinées et que les communications faites ont une largeur de 64 bits.

CHAPITRE 6 CONCLUSION

6.1 Synthèse des travaux

Nous avons accompli tous les objectifs prévus par ce travail. Nous avons d'abord présenté une méthodologie de caractérisation automatisée de la performance des communications atteignable par des coprocesseurs et appliquée celle-ci à un FPGA Zynq-7020. Nous avons aussi montré qu'en combinant les résultats de cette caractérisation à une analyse statique des communications effectuées par un coprocesseur, il est possible d'obtenir une bonne estimation de la latence des communications de celui-ci en une à deux secondes. Il est ensuite possible de combiner cette latence des communications à la latence de la micro-architecture du coprocesseur pour estimer le temps d'exécution total de ce dit coprocesseur en quelques dizaines de secondes.

Pour réaliser la méthodologie de caractérisation automatisée de la performance des communications, nous avons d'abord déterminé les facteurs influençant cette performance en se basant sur un mélange de notions vues dans la littérature et d'expérimentation. Nous avons ensuite développé cette méthodologie en visant le juste milieu entre un temps de caractérisation raisonnable et une précision des résultats assez bonne pour pouvoir estimer correctement toute communication pouvant être faite. Nous avons ensuite appliqué cette méthodologie au Zynq-7020 à l'aide de deux cartes de développement *Zedboard*, qui ont permis une caractérisation automatisée de la performance des communications de ce FPGA-SoC en 2 semaines.

Pour analyser les communications qu'un coprocesseur peut faire avec le reste du système, nous avons développé *HLSComms*, un outil d'analyse statique de code C/C++ synthétisable basé sur LLVM. Cet outil analyse correctement les communications effectuées si le nombre d'itérations des boucles est connu à la compilation. Dans le cas contraire, il demande une rétroaction à l'utilisateur.

Finalement, nous avons développé une procédure d'estimation statique de la latence d'exécution de l'implémentation d'un coprocesseur sur un FPGA donné. Celle-ci est obtenue en combinant une estimation de la latence des communications et de la micro-architecture de ce coprocesseur. La latence des communications est obtenue en faisant correspondre les communications effectuées par le coprocesseur aux données de la caractérisation automatisée du FPGA. La latence de la micro-architecture est fournie par l'outil HLS. Pour les 6 cas de tests effectués sur différents chemins des données du Zynq-7020, nous obtenons une erreur d'estimation moyenne de 9% et maximale de 25%.

6.2 Limitations de la solution proposée

HLSComms est présentement limité par les points présentés à la section 4.2.9 du chapitre 4. Comme nous l'avions mentionné, ces limitations ne sont dues qu'à un manque de temps et seront contournées dans des travaux futurs. La limitation la plus importante est l'absence de support de fonctions membres de classes, qui réduit grandement le nombre d'applications C++ pouvant être analysées.

L'estimation de la latence des communications présentée au chapitre 5 comporte plusieurs limitations. D'abord, nous modélisons les communications d'un seul coprocesseur avec la mémoire et un processeur général, sans que le reste du système (incluant le processeur général) n'utilise significativement le contrôleur mémoire de la puce. La performance réelle pourrait donc varier significativement si d'autres composantes du système cherchaient à accéder à la mémoire en même temps.

Nous ne modélisons pas non plus les communications entre deux coprocesseurs. Or, des configurations de "pipeline au niveau des tâches", ou le résultat d'un coprocesseur est fourni en entrée à un autre coprocesseur, sont fréquentes dans des systèmes réels. La modélisation de ces interactions amènerait cependant d'autres difficultés, puisqu'il faut avoir connaissance des latences d'exécution de tous les coprocesseurs de ce pipeline pour trouver le goulot d'étranglement et donc la performance réellement atteignable.

Notre approche a aussi comme précondition d'être capable de déterminer par analyse statique le nombre d'itérations de chaque boucle du code du coprocesseur analysé. Tant *HLSComms* que les outils de synthèse de haut niveau peuvent identifier les boucles où cette détermination n'est pas possible afin de faciliter une rétroaction par l'utilisateur. Cependant, même avec cette rétroaction, il n'est pas toujours possible de déterminer ce nombre d'itérations, par exemple si celui-ci dépend de paramètres d'entrée du coprocesseur.

6.3 Travaux futurs

Plusieurs nouveaux travaux pourraient découler de la solution présentée dans ce travail. D'abord, l'analyse statique de la latence d'exécution de coprocesseurs pourrait être intégrée dans un flot de développement d'un système logiciel/matériel. De plus, chacune des composantes de ce travail, soit l'estimation de la latence d'exécution, le processus de caractérisation automatisée de la performance des communications et l'analyse statique des communications, pourraient être améliorés.

6.3.1 Intégration de la solution dans un flot de développement système

L'estimateur statique de latence des communications développé dans ce travail devrait être intégré à un flot de développement d'un système logiciel/matériel complet tel que présenté à la figure 6.1. Ce flot, qui prend en entrée le code du système (en C/C++) et les requis de performance, surface et puissance consommée, est similaire à des approches de codesign existantes. Il s'appuierait cependant sur notre analyse statique de la performance coprocesseurs plutôt que sur une analyse dynamique de ceux-ci.

Ce flot serait séparé en 4 étapes. La première consisterait en l'analyse de la performance du système. Celle-ci combinerait l'estimation statique de la latence d'exécution de toutes les fonctions (ou tâches) pouvant potentiellement être exécutées en matériel et un profilage dynamique de l'exécution du système complet sur le processeur général cible (ou un simulateur). Ceci permettrait de connaître la performance de la version logicielle et matérielle de toutes les tâches ciblées. Ces résultats seraient obtenus beaucoup plus rapidement qu'avec les approches de co-simulation existantes puisque les interactions entre les coprocesseurs et le logiciel n'ont pas à être simulés. Cette étape permettrait aussi une vérification de l'exécution correcte de la version logicielle du système. Comme la version matérielle de ces tâches utilise le même code source, nous pouvons aussi penser avec un grand niveau de confiance que l'exécution matérielle serait aussi correcte.

La deuxième étape consiste en l'utilisation des résultats du profilage pour assigner chaque tâche à une implémentation logicielle ou matérielle. Pour les implémentations matérielles, un chemin des données vers la mémoire (si la plateforme utilisée en supporte plusieurs) est aussi sélectionné à cette étape. Si la caractérisation de plusieurs modèles de FPGA et FPGA-SoC est disponible, il serait aussi possible de choisir le modèle le plus approprié.

La troisième étape consiste en la modification du code des coprocesseur pour améliorer leur architecture. Comme nous l'avions vu à la section 2.5 de la revue des concepts, bien que la synthèse de haut niveau permette d'utiliser directement le code C/C++ d'une application logicielle, le résultat est rarement optimal et peut généralement être énormément amélioré. À ce stade, le code résultant des modifications pourrait être profilé à l'aide de notre analyseur statique et validé à l'aide de simulations C/C++ ou profilé et validé par une co-simulation logicielle/matérielle. Finalement, une fois les requis satisfaits, le système est implémenté et validé.

Finalement, le flot présenté ici suppose que les tâches (ou fonctions) dont la performance de la version coprocesseur doit être profilée statiquement sont choisis à l'avance par l'utilisateur. Cependant, nous pourrions aussi envisager dans un deuxième temps une automatisation

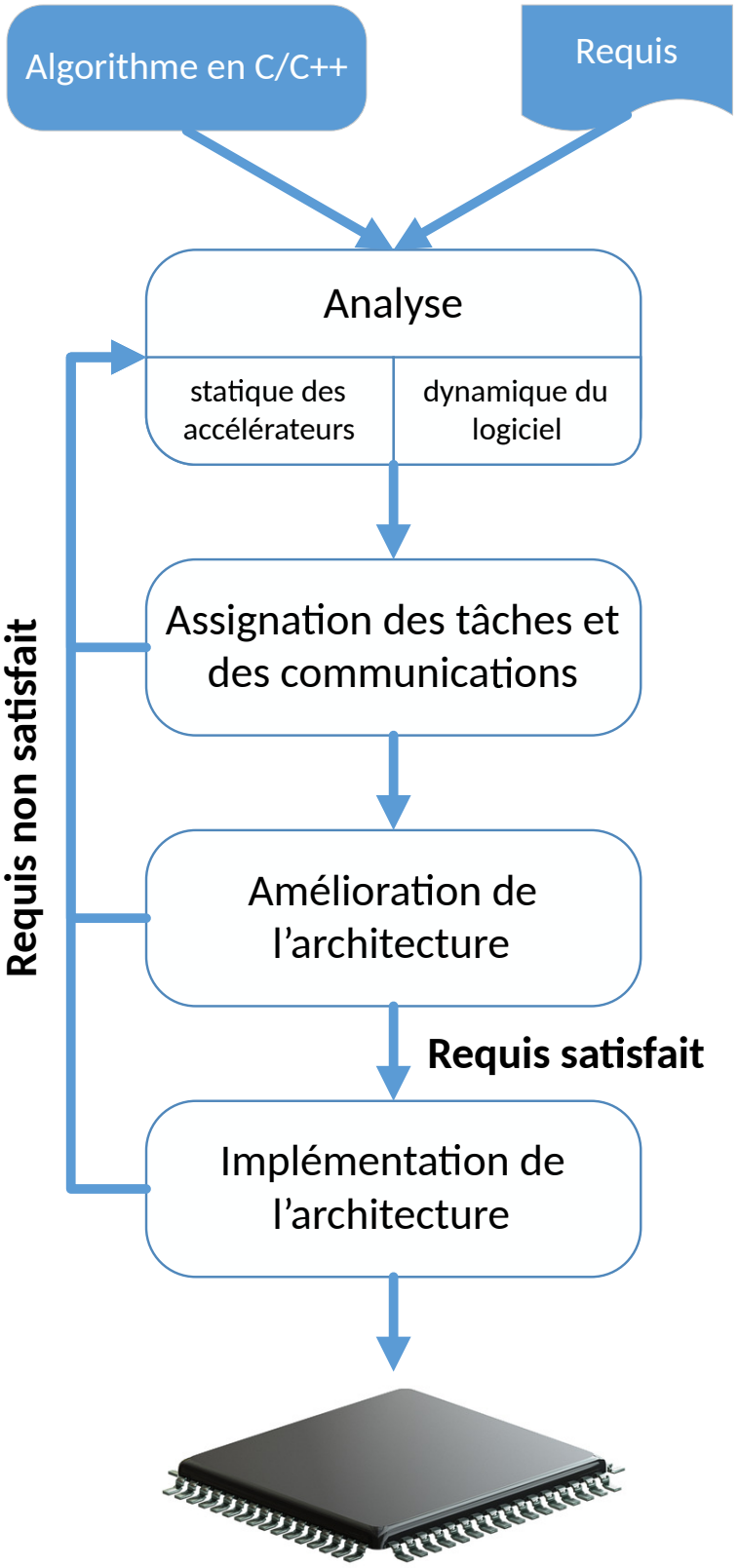


Figure 6.1 Intégration de l'analyse statique de la latence de coprocesseurs dans un flot de développement système

de ce processus en fonction du résultat du profilage dynamique du logiciel. De même, le partitionnement logiciel/matériel et la sélection du meilleur chemin des données pourrait aussi être automatisé. Ceci permettrait de faire un partitionnement logiciel/matériel d'un système de manière rapide et entièrement automatisée.

6.3.2 Travaux futurs pour l'estimation de la latence d'exécution

D'abord et avant tout, l'estimation de la latence d'exécution présentée au chapitre 5 devrait être automatisée. Comme nous l'avions mentionné dans ce chapitre, nous avons omis cette étape par manque de temps, mais ne nous attendons pas à ce que cette automatisation puisse poser de difficultés techniques significatives.

De plus, la procédure d'estimation pourrait être améliorée pour donner un résultat plus représentatif de la réalité, surtout si les boucles ne sont pas pipelinées. Un bon point de départ pourrait être de considérer que bien que l'estimation de la latence de la micro-architecture ne tienne pas compte des délais des communications à l'extérieur du coprocesseur, elle modélise la latence induite par le contrôle des interfaces utilisées par ces communications. Or, cette latence est aussi prise en compte dans le calcul de la latence des communications. Cette situation n'est pas problématique pour des boucles parfaitement pipelinées, puisque utilisions le maximum de la latence des communications ou de la micro-architecture. Cependant, dans les portions à l'extérieur de boucles ou dans des boucles non pipelinées ou pipelinées avec un intervalle d'initiation supérieur à 1 (c-a-d où il faut plus d'un cycle entre chaque nouvelle itération du pipeline), le délai induit par ce contrôle est pris en considération 2 fois. Il serait donc intéressant de tester le remplacement de l'équation 5.1 par :

$$t_{\text{tot}} = \left(\sum_{i=0}^{N} (t_{\mu\text{arch}}(i) - t_{\mu\text{arch comms}}(i)) + t_{\text{comms}}(i) \right) + (t_{\mu\text{arch scal}} - t_{\mu\text{arch comms scal}}) + t_{\text{comms scal}} \tag{6.1}$$

où $\text{t}_{\mu\text{arch comms}}(i)$ et $\text{t}_{\mu\text{arch comms scal}}$ représentent le nombre de cycles où le coprocesseur contrôle ses interfaces de communication. Cette information peut être obtenue en analysant les rapports produits par Vivado HLS.

6.3.3 Travaux futurs pour la méthodologie de caractérisation automatisée des communications

Une avenue intéressante de travaux futurs serait d'étendre la méthodologie de caractérisation automatisée des communications aux communications entre un FPGA et un processeur

général hors puce (par ex. par PCI-E). Ceci permettrait d'étendre notre estimation de la latence d'exécution à des applications destinées aux centres de données.

Outre ces travaux futurs, quelques modifications pourraient aussi être à apportées à la caractérisation automatisée des communications telle que présentée ici. D'abord, si la largeur des transferts sur le chemin des données à caractériser est configurable, toutes les largeurs possibles (par ex. 8, 32 et 64 bits) devraient l'être. Ceci permettrait de caractériser correctement des cas comme le filtre de Sobel non modifié de la section 5.2.1.

Ensuite, nous réduirions le nombre de fois où le même coprocesseur avec les mêmes paramètres est exécuté. Pour caractériser le Zynq-7020, nous avions en effet exécuté chaque cas de test 51200 fois. Ce nombre était arbitraire et représentait 10 exécutions successives d'un programme faisant la moyenne de 1024 exécutions du coprocesseur de test (où le nombre 1024 provenait d'un exemple de Xilinx), exécuté 5 fois de suite avec 5 secondes entre chaque itération. Or, la variation du résultat entre chaque exécution de ce programme était minime, surtout pour les transferts de grande taille, qui prenaient justement la grande majorité du temps d'exécution des tests. Ce nombre pourrait donc être réduit considérablement pour ces grands transferts. Nous croyons que ceci permettrait de réduire de 5 à 10x le temps d'exécution des tests.

Il serait aussi pertinent de chercher d'autres facteurs pouvant affecter la performance des communications entre le coprocesseur et la mémoire. Entre autres, nous n'avons pas testé l'impact que pourrait avoir une puce mémoire plus ou moins rapide.

Finalement, outre la latence des communications, il serait approprié de caractériser l'énergie et les ressources consommées par le moyen de communication employé, ce qui permettrait de prendre des décisions architecturales plus informées. Nous nous attendons en effet à ce que celles-ci varient en fonction du type et nombre de ports de communications utilisé et de la largeur du chemin des données. De plus, l'énergie consommée par octet transféré devrait varier en fonction de la quantité de données transférées (et ce, particulièrement dans le cas de transferts cohérents en cache) [20]. Elle varie aussi probablement en fonction de la présence de rafales ou non dans le transfert.

6.3.4 Travaux futurs pour *HLSComms*

Outre contourner les limitations présentés à la section 4.2.9, nous croyons que la structure logicielle *HLSComms* pourrait être améliorée, de même que sa méthodologie de développement.

Nous modifierions en effet la structure logicielle de l'outil (qui était présentée à la figure 4.2)

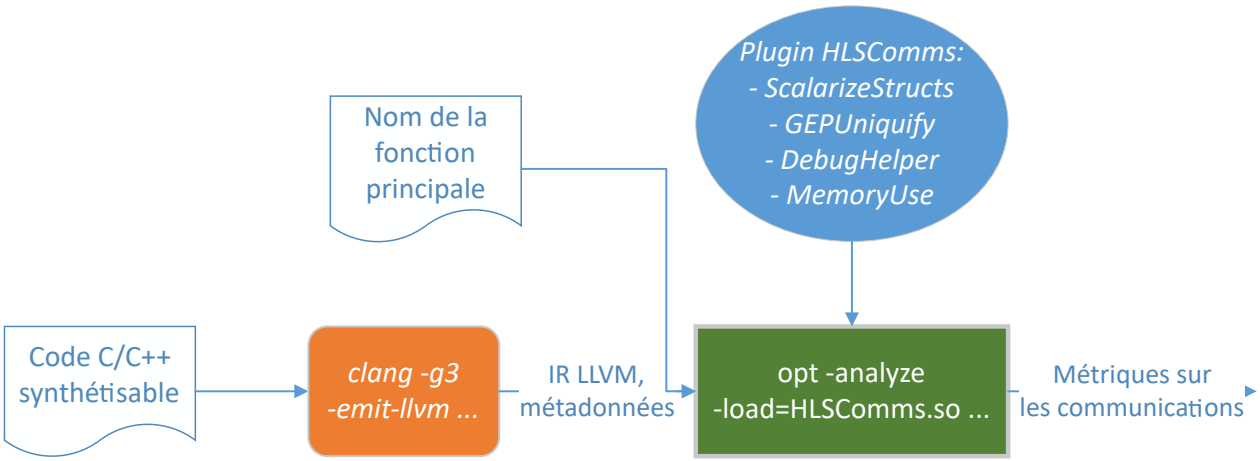


Figure 6.2 Suggestion d'un flot amélioré pour *HLSComms*.

pour passer d'un binaire monolithique à une suite de scripts. Notamment, nous séparerions le prétraitement par Clang et l'analyse effectuée à l'aide de LLVM. Le flot modifié est présenté à la figure 6.2.

La passe *FindFunctions* serait complètement éliminée, puisque toutes les informations obtenues et vérifiées dans cette passe se retrouvent aussi les métadonnées de débogage liées à l'IR. De plus, certaines fonctionnalités implicites de code C++ comme la présence de l'argument *this* pour les fonctions membres de classes ne se retrouvent pas explicitement dans l'AST, mais se trouvent explicitement dans les métadonnées de débogage. Utiliser ces métadonnées serait donc plus simple.

Nous enlèverions aussi *StructScalarizer* pour le remplacer par une analyse et transformation des structures sous la forme d'une nouvelle passe LLVM, *ScalarizeStructs*. Nous croyons que cela simplifierait le support de fonctions membres de classes.

De plus, Clang serait appelé directement par le biais d'un script plutôt que par un exécutable. Finalement, toutes les passes LLVM développées se retrouveraient dans un plugin chargé dynamiquement par l'invocation de l'optimiseur LLVM *opt*. Ce programme, qui est chargé d'appliquer les passes spécifiées au fichier source d'entrée puis optionnellement d'écrire le résultat, permet le chargement dynamique de modules d'extension contenant des passes compilées séparément.

La méthodologie de développement pourrait aussi être revue. Nous croyons que l'approche utilisée, qui consistait à écrire quelques tests simples en C/C++, générer l'IR et vérifier le résultat de l'analyse est bonne pour commencer le développement. Cependant, nous croyons être arrivé à un point où cette approche devrait être remplacée par des tests écrits en IR LLVM

directement. Plutôt que de tester plusieurs manières différentes de faire des communications en C/C++, nous devrions nous concentrer à gérer tous les cas possibles de l'IR. Nous croyons qu'une telle approche limiterait les surprises rencontrées en testant de nouveaux cas.

RÉFÉRENCES ET BIBLIOGRAPHIE

RÉFÉRENCES

[1] Xilinx, *Vivado Design Suite User Guide : High-Level Synthesis (UG902)*, 2017.4, déc. 2017. adresse : `www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf`.

[2] É. Gauthier, « Exploration d'une méthodologie de développement matériel et logiciel au niveau système appliqué à un système d'encodage de flux vidéo évolutif », mém. de mast., Université de Montréal, juil. 2017.

[3] J. Schlessman, C.-Y. Chen, W. Wolf, B. Ozer, K. Fujino et K. Itoh, « Hardware/Software Co-design of an FPGA-based Embedded Tracking System », in *Computer Vision and Pattern Recognition Workshop, 2006. CVPRW'06. Conference on*, IEEE, 2006, p. 123-123.

[4] C.-C. Lin, J.-W. Chen, H.-C. Chang, Y.-C. Yang, Y.-H. O. Yang, M.-C. Tsai, J.-I. Guo et J.-S. Wang, « A 160K Gates/4.5 KB SRAM H.264 Video Decoder for HDTV Applications », *IEEE Journal of Solid-State Circuits*, t. 42, n° 1, p. 170-182, 2007.

[5] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song et al., « Going Deeper with Embedded FPGA Platform for Convolutional Neural Network », in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2016, p. 26-35.

[6] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray et al., « A reconfigurable fabric for accelerating large-scale datacenter services », in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, IEEE, 2014, p. 13-24.

[7] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan et M. Margala, « An FPGA Memcached Appliance », in *Proceedings of the ACM/-SIGDA international symposium on Field programmable gate arrays*, ACM, 2013, p. 245-254.

[8] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers et al., « In-Datacenter Performance Analysis of a Tensor Processing Unit », in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ACM, juin 2017, p. 1-12.

[9] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim et al., « A Cloud-Scale Acceleration Architecture », in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, IEEE, 2016, p. 1-13.

[10] W. A. Wulf et S. A. McKee, « Hitting the Memory Wall : Implications of the Obvious », *ACM SIGARCH computer architecture news*, t. 23, nº 1, p. 20-24, 1995.

[11] S. A. McKee, « Reflections on the Memory Wall », in *Proceedings of the 1st conference on Computing frontiers*, ACM, 2004, p. 162.

[12] J. L. Hennessy et D. A. Patterson, *Computer Architecture : a Quantitative Approach*, 5e éd. Elsevier, 2011.

[13] S. Williams, A. Waterman et D. Patterson, « Roofline : An Insightful Visual Performance Model for Multicore Architectures », *Communications of the ACM*, t. 52, nº 4, p. 65-76, 2009.

[14] B. da Silva, A. Braeken, E. H. D'Hollander et A. Touhafi, « Performance Modeling for FPGAs : Extending the Roofline Model with High-Level Synthesis Tools », *International Journal of Reconfigurable Computing*, t. 2013, p. 7, 2013.

[15] M. Naylor, P. J. Fox, A. T. Markettos et S. W. Moore, « Managing the FPGA Memory Wall : Custom Computing or Vector Processing ? », in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, IEEE, 2013, p. 1-6.

[16] Xilinx, *Zynq-7000 all programmable soc technical reference manual (UG585)*, 1.12.1, déc. 2017. adresse : `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`.

[17] S. W. Nabi et W. Vanderbauhede, « FPGA Design Space Exploration for Scientific HPC Applications Using a Fast and Accurate Cost Model Based on Roofline Analysis », *Journal of Parallel and Distributed Computing*, 2017.

[18] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao et J. Cong, « Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks », in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2015, p. 161-170.

[19] A. Desaulty, « Intégration d'un outil de traduction assistée de code C pour la création de coprocesseurs matériels dans un flot de conception système », mém. de mast., Université de Montréal, 2016.

[20] M. Sadri, C. Weis, N. Wehn et L. Benini, « Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ », in *Proceedings of the 10th FPGAworld Conference*, ACM, 2013, p. 5.

[21] M. Göbel, A. Elhossini, C. C. Chi, M. Alvarez-Mesa et B. Juurlink, « A Quantitative Analysis of the Memory Architecture of FPGA-SoCs », in *Applied Reconfigurable Computing : 13th International Symposium, ARC 2017, Delft, The Netherlands, April 3-7, 2017, Proceedings*, S. Wong, A. C. Beck, K. Bertels et L. Carro, éd. Cham : Springer International Publishing, 2017, p. 241-252, isbn : 978-3-319-56258-2. doi : 10.1007/978-3-319-56258-2_21. adresse : http://dx.doi.org/10.1007/978-3-319-56258-2_21.

[23] Intel FPGA, *Cyclone v hard processor system technical reference manual*, cv_5v4, oct. 2016. adresse : https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cv_5v4.pdf.

[24] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia et al., « A Reconfigurable Computing System Based on a Cache-Coherent Fabric », in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, IEEE, 2011, p. 80-85.

[25] J. Stuecheli, B. Blaner, C. Johns et M. Siegel, « CAPI : A Coherent Accelerator Processor Interface », *IBM Journal of Research and Development*, t. 59, n° 1, p. 7-1, 2015.

[26] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman et P. Wei, « A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms », in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, IEEE, 2016, p. 1-6.

[28] Xilinx, *Zynq ultrascale+ device technical reference manual (UG1085)*, 1.7, déc. 2017. adresse : https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf.

[29] Intel FPGA, *Stratix 10 GX/SX Device Overview*, oct. 2017. adresse : https://www.altera.com/en_US/pdfs/literature/hb/stratix-10/s10-overview.pdf.

[30] Y. Hao, Z. Fang, G. Reinman et J. Cong, « Supporting Address Translation for Accelerator-Centric Architectures », in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, IEEE, 2017, p. 37-48.

[31] P. Vogel, A. Kurth, J. Weinbuch, A. Marongiu et L. Benini, « Efficient Virtual Memory Sharing via On-Accelerator Page Table Walking in Heterogeneous Embedded SoCs », *ACM Transactions on Embedded Computing Systems (TECS)*, t. 16, nº 5s, p. 154, 2017.

[33] J. Silva, V. Sklyarov et I. Skliarova, « Comparison of on-chip communications in Zynq-7000 all programmable systems-on-chip », *IEEE Embedded Systems Letters*, t. 7, nº 1, p. 31-34, 2015.

[34] H. Ding et M. Huang, « Improve memory access for achieving both performance and energy efficiencies on heterogeneous systems », in *Field-Programmable Technology (FPT), 2014 International Conference on*, IEEE, 2014, p. 91-98.

[35] M. Tahghighi, S. Sinha et W. Zhang, « Analytical Delay Model for CPU-FPGA Data Paths in Programmable System-on-Chip FPGA », in *Applied Reconfigurable Computing : 12th International Symposium, ARC 2016 Mangaratiba, RJ, Brazil, March 22–24, 2016 Proceedings*, V. Bonato, C. Bouganis et M. Gorgon, éd. Cham : Springer International Publishing, 2016, p. 159-170, isbn : 978-3-319-30481-6. doi : 10.1007/978-3-319-30481-6_13. adresse : http://dx.doi.org/10.1007/978-3-319-30481-6_13.

[37] R. Nikhil, « Bluespec System Verilog : Efficient, Correct RTL from High-Level Specifications », in *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, IEEE, 2004, p. 69-70.

[38] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek et K. Asanović, « Chisel : Constructing Hardware in a Scala Embedded Language », in *Proceedings of the 49th Annual Design Automation Conference*, ACM, 2012, p. 1216-1225.

[39] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras et D. P. Singh, « From OpenCL to High-Performance Hardware on FPGAs », in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, IEEE, 2012, p. 531-534.

[40] *HDL Coder - MATLAB & Simulink*. adresse : https://fr.mathworks.com/products/hdl-coder.html.

[41] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal et M. Luján, « An empirical evaluation of high-level synthesis languages and tools for database acceleration », in *Field Pro-*

grammable Logic and Applications (FPL), 2014 24th International Conference on, IEEE, 2014, p. 1-8.

[42] G. Wang, H. Lam, A. George et G. Edwards, « Performance and Productivity Evaluation of Hybrid-Threading HLS versus HDLs », in *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, IEEE, 2015, p. 1-7.

[43] F. Hannig, « A Quick Tour of High-Level Synthesis Solutions for FPGAs », in *FPGAs for Software Programmers*, Springer, 2016, p. 49-59.

[44] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi et othersnane, « A Survey and Evaluation of FPGA High-Level Synthesis Tools », *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2015.

[45] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown et T. Czajkowski, « LegUp : High-Level Synthesis for FPGA-based Processor/Accelerator Sviystems », in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ACM, 2011, p. 33-36.

[46] L. Moss, H. Guerard, G. Dare et G. Bois, « An ESL Methodology for Rapid Creation of Embedded Aerospace Systems using Hardware-Software Co-design on Virtual Platforms », SAE Technical Paper, rapp. tech., 2012.

[48] L. Computing et Inc., *LegUp User Manual Release*, 5.1, juil. 2017. adresse : `https://www.legupcomputing.com/docs/legup-5.1-docs/legup-5.1-docs.pdf`.

[49] Y. Hara, H. Tomiyama, S. Honda et H. Takada, « Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis », *Journal of Information Processing*, t. 17, p. 242-254, 2009.

[50] *The LLVM Compiler Infrastructure*. adresse : `https://llvm.org/`.

[51] C. Lattner et V. Adve, « LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation », in *Proceedings of the international symposium on Code generation and optimization : feedback-directed and runtime optimization*, IEEE Computer Society, 2004, p. 75.

[52] C. Lattner, « LLVM and Clang : Next Generation Compiler Technology », in *The BSD Conference*, 2008, p. 1-2.

[53] *clang : a C language family frontend for LLVM*. adresse : `https://clang.llvm.org/`.

[54] H. K. Wright, D. Jasper, M. Klimek, C. Carruth et Z. Wan, « Large-Scale Automated Refactoring Using ClangMR. », in *ICSM*, 2013, p. 548-551.

[55] *Clang Static Analyzer*. adresse : `https://clang-analyzer.llvm.org/`.

[56] *Clangd - Extra Clang Tools 7 documentation*. adresse : `https://clang.llvm.org/extra/clangd.html`.

[57] *LLVM Language Reference Manual*. adresse : `https://releases.llvm.org/4.0.1/docs/LangRef.html`.

[58] N. Lewycky, *The Use of getSmallConstantTripCount Method of Loop in LLVM*. adresse : `https://stackoverflow.com/questions/5809481/the-use-of-getsmallconstanttripcount-method-of-loop-in-llvm`.

[59] O. Bachmann, P. S. Wang et E. V. Zima, « Chains of Recurrences — a method to expedite the evaluation of closed-form functions », in *Proceedings of the international symposium on Symbolic and algebraic computation*, ACM, 1994, p. 242-249.

[60] D. Quinlan, « ROSE : Compiler Support for Object-oriented Frameworks », *Parallel Processing Letters*, t. 10, n° 02n03, p. 215-226, 2000.

[61] A. Cilardo et L. Gallo, « Improving Multibank Memory Access Parallelism with Lattice-based Partitioning », *ACM Transactions on Architecture and Code Optimization (TACO)*, t. 11, n° 4, p. 45, 2015.

[62] L.-N. Pouchet, P. Zhang, P. Sadayappan et J. Cong, « Polyhedral-based Data Reuse Optimization for Configurable Computing », in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, ACM, 2013, p. 29-38.

[63] C. Bastoul, « Code generation in the polyhedral model is easier than you think », in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, IEEE Computer Society, 2004, p. 7-16.

[64] H.-J. Yang, K. Fleming, M. Adler, F. Winterstein et J. Emer, « Scavenger : Automating the Construction of Application-Optimized Memory Hierarchies », in *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, IEEE, 2015, p. 1-8.

[65] J. Liu, J. Wickerson et G. A. Constantinides, « Loop Splitting for Efficient Pipelining in High-Level Synthesis », in *IEEE Int. Symp. on Field-Programmable Custom Computing Machines (FCCM '16)*, 2016.

[66] M. Alle, A. Morvan et S. Derrien, « Runtime Dependency Analysis for Loop Pipelining in High-Level Synthesis », English, in *50TH ACM / EDAC / IEEE DESIGN AUTOMATION CONFERENCE (DAC)*, sér. Design Automation Conference DAC, ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, MAY 29-JUN 07, 2013, ACM ; EDAC ; IEEE, IEEE COMPUTER SOC, 2013, isbn : 978-1-4503-2071-9.

[67] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen et J. Cong, « Improving high level synthesis optimization opportunity through polyhedral transformations », in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, ACM, 2013, p. 9-18.

[68] F. Winterstein, « Separation Logic for High-level Synthesis », thèse de doct., Imperial College London, 2017. doi : 10.1007/978-3-319-53222-6.

[69] J. Cong, M. Huang, P. Pan, Y. Wang et P. Zhang, « Source-to-Source Optimization for HLS », in *FPGAs for Software Programmers*, D. Koch, F. Hannig et D. Ziener, éd. Cham : Springer International Publishing, 2016, p. 137-163, isbn : 978-3-319-26408-0. doi : 10.1007/978-3-319-26408-0_8. adresse : http://dx.doi.org/10.1007/978-3-319-26408-0_8.

[70] F. Ghenassia et al., *Transaction-level modeling with SystemC*. Springer, 2005.

[71] *Projects built with LLVM*. adresse : https://llvm.org/ProjectsWithLLVM/.

[72] *LLVM Related Publications*. adresse : https://llvm.org/pubs/.

[73] *libgccjit*, fév. 2017. adresse : https://gcc.gnu.org/onlinedocs/jit/.

[74] *JSON for Modern C++*. adresse : https://github.com/nlohmann/json.

[75] *Boost C++ Libraries*. adresse : http://www.boost.org/.

[76] Xilinx, *SDSoC Environment Optimization Guide (UG1235)*, 2017.4, déc. 2017. adresse : www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1235-sdsoc-optimization-guide.pdf.

[77] ——, *UltraFast High-Level Productivity Design Methodology Guide (UG1197)*, 2017.4, déc. 2017. adresse : https://www.xilinx.com/support/documentation/sw_manuals/ug1197-vivado-high-level-productivity.pdf.

[78] *CHStone Home Page*. adresse : http://www.ertl.jp/chstone/.

BIBLIOGRAPHIE

M. Alipour, M. E. Salehi et K. Moshari, « Cache power and performance tradeoffs for embedded applications », in *Computer Applications and Industrial Electronics (ICCAIE), 2011 IEEE International Conference on*, IEEE, 2011, p. 26-31.

B. Bosi, G. Bois et Y. Savaria, « Reconfigurable pipelined 2-D convolvers for fast digital signal processing », *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, t. 7, n° 3, p. 299-308, 1999.

J. Cardoso et M. Hübner, *Reconfigurable Computing : From FPGAs to Hardware/-Software Codesign*. Springer Science & Business Media, 2011.

A. Cilardo, P. Durante, C. Lofiego et A. Mazzeo, « Early Prediction of Hardware Complexity in HLL-to-HDL Translation », in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, IEEE, 2010, p. 483-488.

A. Cilardo, L. Gallo et N. Mazzocca, « Design Space Exploration for High-level Synthesis of Multi-threaded Applications », *Journal of Systems Architecture*, t. 59, n° 10, p. 1171-1183, 2013.

P. Coussy, D. D. Gajski, M. Meredith et A. Takach, « An introduction to High-Level Synthesis », *IEEE Design & Test of Computers*, t. 26, n° 4, p. 8-17, 2009.

H. Giefers, P. Staar et R. Polig, « Energy-Efficient Stochastic Matrix Function Estimator for Graph Analytics on FPGA », in *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, IEEE, 2016, p. 1-9.

S. Lafond et J. Lilius, « Interrupt costs in embedded system with short latency hardware accelerators », in *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, IEEE, 2008, p. 317-325.

Z. Li, A. Jannesari et F. Wolf, « Discovery of potential parallelism in sequential programs », in *2013 42nd International Conference on Parallel Processing*, IEEE, 2013, p. 1004-1013.

Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do et D. Chen, « High-level synthesis : productivity, performance, and software constraints », *Journal of Electrical and Computer Engineering*, t. 2012, p. 1, 2012.

D. Liu et B. C. Schafer, « Efficient and reliable High-Level Synthesis Design Space Explorer for FPGAs », in *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, EPFL, 2016, p. 1-8.

C. Lo et P. Chow, « Model-Based Optimization of High Level Synthesis Directives », English, in *26TH INTERNATIONAL CONFERENCE ON FIELD PROGRAMMABLE LOGIC AND APPLICATIONS (FPL)*, IEEE, 2016.

S. Madelénat, C. Labreuche, J. Le Noir et G. Gailliard, « Comparing several candidate architectures variants : An Industrial Case Study », in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

A. Mazaheri, A. Jannesari, A. Mirzaei et F. Wolf, « Characterizing Loop-Level Communication Patterns in Shared Memory », in *Parallel Processing (ICPP), 2015 44th International Conference on*, IEEE, 2015, p. 759-768.

R. Murphy, « On the Effects of Memory Latency and Bandwidth on Supercomputer Application Performance », in *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, IEEE, 2007, p. 35-43.

C. Pilato, P. Mantovani, G. Di Guglielmo et L. P. Carloni, « System-Level Memory Optimization for High-Level Synthesis of Component-Based SoCs », in *Hardware/-Software Codesign and System Synthesis (CODES+ ISSS), 2014 International Conference on*, IEEE, 2014, p. 1-10.

O. Reiche, M. Schmid, F. Hannig, R. Membarth et J. Teich, « Code generation from a domain-specific language for C-based HLS of hardware accelerators », in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2014 International Conference on*, IEEE, 2014, p. 1-10.

B. C. Schafer et K. Wakabayashi, « Machine-Learning Predictive Modelling High-Level Synthesis Design Space Exploration », English, *IET COMPUTERS AND DIGITAL TECHNIQUES*, t. 6, n° 3, 153-159, mai 2012, ISSN : 1751-8601. DOI : {10.1049/iet-cdt.2011.0115}.

K. Shagrithaya, K. Kępa et P. Athanas, « Enabling development of OpenCL applications on FPGA platforms », in *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, IEEE, 2013, p. 26-30.

P. Vogel, A. Marongiu et L. Benini, « An Evaluation of Memory Sharing Performance for Heterogeneous Embedded SoCs with Many-Core Accelerators », in *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores*, ACM, 2015, p. 6.

F. Winterstein, S. Bayliss et G. A. Constantinides, « Separation logic-assisted code transformations for efficient high-level synthesis », in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, IEEE, 2014, p. 1-8.

Xilinx, *SDSoC Environment User Guide (UG1027)*, 2017.4, déc. 2017. adresse : `www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1027-sdsoc-user-guide.pdf`.

——, *Vivado Design Suite : AXI Reference Guide (UG1037)*, 4.0, juil. 2017. adresse : `www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf`.

——, *Vivado HLS : Improving Performance*, 2013.3, 2013. adresse : `https://forums.xilinx.com/xlnx/attachments/xlnx/hls/6473/1/VivadoHLS_Improving_Performance.pdf`.

Falcon Computing - Acceleration Simplified. adresse : `https://www.falcon-computing.com/`.

SDSoC Development Environment, `https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html`. adresse : `https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html`.

D. Gshwend, « ZynqNet : An FPGA-Accelerated Embedded Convolutional Neural Network », mém. de mast., ETH Zürich, août 2016. adresse : `https://github.com/dgschwend/zynqnet`.

J. Cong, P. Wei, C. H. Yu et P. Zhou, « Bandwidth Optimization Through On-Chip Memory Restructuring for HLS », in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*, IEEE, 2017, p. 1-6.

AMD, *AMD I/O Virtualization Technology (IOMMU) Specification*, 3.00, 2016.

Intel, *Intel® Virtualization Technology for Directed I/O Architecture Specification*, nov. 2017.

A. V. Aho, R. Sethi et J. D. Ullman, *Compilers : Principles, Techniques, and Tools*. Addison-wesley Reading, 2007, t. 2.

ANNEXE A MODIFICATIONS APPORTÉES AUX BANCS DE TESTS

Cette annexe présente explicitement les modifications apportées aux cas de tests utilisés pour démontrer la faisabilité de l'analyse statique des communications par rapport à leur source originale.

CHStone

Cette section représente les modifications apportées à CHStone pour réaliser les tests d'*HLSComms*. Elles sont basées sur la version 1.11 du banc de test, qui peut être obtenu à [78].

Ces modifications ont principalement servi à s'assurer que les bancs de tests recevaient par paramètre pointeur les données qu'ils ont à traiter plutôt qu'avoir directement accès à un tableau les contenant comme variables globales. Notons que la directive *#pragma clang loop interleave_count* a aussi été ajouté à certaines boucles. Celle-ci sert à indiquer à *HLSComms* la quantité maximale d'itérations que la boucle peut prendre. Finalement, quelques modifications mineures peuvent avoir été apportées au reste du code pour contourner des limitations d'*HLSComms*.

ADPCM

```
--- a/adpcm/adpcm.c
+++ b/adpcm/adpcm.c
@@ -834,7 +834,7 @@ const int test_result[SIZE] = {
 };

 void
-adpcm_main ()
+adpcm_main (const int* test_data_ ,int* result_ /* out*/)
 {
   int i, j;

@@ -845,13 +845,13 @@ adpcm_main ()

   for (i = 0; i < IN_END; i += 2)
     {
-      compressed[i / 2] = encode (test_data[i], test_data[i + 1]);
+      compressed[i / 2] = encode (test_data_[i], test_data_[i + 1]);
     }
   for (i = 0; i < IN_END; i += 2)
```

```
     {
       decode (compressed[i / 2]);
-      result[i] = xout1;
-      result[i + 1] = xout2;
+      result_[i] = xout1;
+      result_[i + 1] = xout2;
     }
 }

@@ -862,7 +862,7 @@ main ()
   int main_result;

       main_result = 0;
-      adpcm_main ();
+      adpcm_main (test_data, result);
       for (i = 0; i < IN_END / 2; i++)
         {
           if (compressed[i] != test_compressed[i])
```

Figure A.1 Modifications apportées à adpcm.c pour réaliser les tests d'*HLSComms*.

AES

```
--- a/aes/aes.h
+++ b/aes/aes.h
@@ -61,9 +61,7 @@
  */

 /* ************* data type define ************************* */
-int type;
 int nb;
-int round_val;
 int key[32];
 int statemt[32];
 int word[4][120];
diff --git a/aes/aes_dec.c b/aes/aes_dec.c
index 25173f3..c585247 100644
--- a/aes/aes_dec.c
+++ b/aes/aes_dec.c
@@ -59,10 +59,14 @@
  *   WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
  *
  */
+#include "aes.h"
+#include "aes_key.c"
```

```
+#include "aes_func.c"
+
 int
 decrypt (int statemt[32], int key[32], int type)
 {
-  int i;
+  int i, round_val;
 /*
 +-----------------------------------------------------------------------+
 | * Test Vector (added for CHStone)                                     |
@@ -113,6 +117,7 @@ decrypt (int statemt[32], int key[32], int type)

   InversShiftRow_ByteSub (statemt, nb);

+#pragma clang loop interleave_count(12)
   for (i = round_val - 1; i >= 1; --i)
     {
       AddRoundKey_InversMixColumn (statemt, nb, i);
@@ -121,16 +126,5 @@ decrypt (int statemt[32], int key[32], int type)

   AddRoundKey (statemt, type, 0);

-  printf ("\ndecrypto message\t");
-  for (i = 0; i < ((type % 1000) / 8); ++i)
-    {
-      if (statemt[i] < 16)
-        printf ("0");
-      printf ("%x", statemt[i]);
-    }
-
-  for (i = 0; i < 16; i++)
-    main_result += (statemt[i] != out_dec_statemt[i]);
-
   return 0;
 }
--- a/aes/aes_enc.c
+++ b/aes/aes_enc.c
@@ -59,11 +59,14 @@
  *   WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
  *
  */
+#include "aes.h"
+#include "aes_key.c"
+#include "aes_func.c"
 /*  ******* encrypto ************ */
```

```
 int
 encrypt (int statemt[32], int key[32], int type)
 {
-  int i;
+  int i, round_val;
 /*
 +--------------------------------------------------------------------------+
 | * Test Vector (added for CHStone)                                        |
@@ -115,16 +118,5 @@ encrypt (int statemt[32], int key[32], int type)
   ByteSub_ShiftRow (statemt, nb);
   AddRoundKey (statemt, type, i);

-  printf ("encrypted message \t");
-  for (i = 0; i < nb * 4; ++i)
-    {
-      if (statemt[i] < 16)
-        printf ("0");
-      printf ("%x", statemt[i]);
-    }
-
-  for (i = 0; i < 16; i++)
-    main_result += (statemt[i] != out_enc_statemt[i]);
-
   return 0;
 }
--- a/aes/aes_func.c
+++ b/aes/aes_func.c
@@ -369,6 +369,7 @@ MixColumn_AddRoundKey (int statemt[32], int nb, int n)
   int ret[8 * 4], j;
   register int x;

+  #pragma clang loop interleave_count(8)
   for (j = 0; j < nb; ++j)
     {
       ret[j * 4] = (statemt[j * 4] << 1);
@@ -419,6 +420,7 @@ MixColumn_AddRoundKey (int statemt[32], int nb, int n)
       ret[3 + j * 4] ^=
         statemt[1 + j * 4] ^ statemt[2 + j * 4] ^ word[3][j + nb * n];
     }
+  #pragma clang loop interleave_count(8)
   for (j = 0; j < nb; ++j)
     {
       statemt[j * 4] = ret[j * 4];
@@ -435,7 +437,7 @@ AddRoundKey_InversMixColumn (int statemt[32], int nb)
 {
```

```
   int ret[8 * 4], i, j;
   register int x;
-
+  #pragma clang loop interleave_count(32)
   for (j = 0; j < nb; ++j)
     {
       statemt[j * 4] ^= word[0][j + nb * n];
@@ -443,6 +445,7 @@ AddRoundKey_InversMixColumn (int statemt[32], int nb)
       statemt[2 + j * 4] ^= word[2][j + nb * n];
       statemt[3 + j * 4] ^= word[3][j + nb * n];
     }
+    #pragma clang loop interleave_count(32)
   for (j = 0; j < nb; ++j)
     for (i = 0; i < 4; ++i)
       {
@@ -497,6 +500,7 @@ AddRoundKey_InversMixColumn (int statemt[32], int nb)
         x ^= statemt[(i + 3) % 4 + j * 4];
         ret[i + j * 4] ^= x;
       }
+#pragma clang loop interleave_count(32)
   for (i = 0; i < nb; ++i)
     {
       statemt[i * 4] = ret[i * 4];
@@ -531,6 +535,7 @@ AddRoundKey (int statemt[32], int type, int n)
       nb = 8;
       break;
     }
+  #pragma clang interleave_count(8)
   for (j = 0; j < nb; ++j)
     {
       statemt[j * 4] ^= word[0][j + nb * n];
--- a/aes/aes_key.c
+++ b/aes/aes_key.c
@@ -129,6 +129,7 @@ KeySchedule (int type, int key[32])
     default:
       return -1;
     }
+#pragma clang loop interleave_count(8)
   for (j = 0; j < nk; ++j)
     for (i = 0; i < 4; ++i)
 /* 0 word */
```

Figure A.2 Modifications apportées à aes pour réaliser les tests d'*HLSComms* sur encrypt et decrypt.

Blowfish

```
--- a/blowfish/bf.c
+++ b/blowfish/bf.c
@@ -814,7 +814,7 @@ const unsigned char out_key[KEYSIZE] = {
 #define N 40

 int
-blowfish_main ()
+blowfish_main (const unsigned char* in_key_)
 {
   unsigned char ukey[8];
   unsigned char indata[N];
@@ -837,30 +837,16 @@ blowfish_main ()
     }
   BF_set_key (8, ukey);
   i = 0;
+  #pragma clang loop interleave_count(KEYSIZE/N)
   while (k < KEYSIZE)
     {
+      #pragma clang loop interleave_count(N)
       while (k < KEYSIZE && i < N)
-        indata[i++] = in_key[k++];
+        indata[i++] = in_key_[k++];

       BF_cfb64_encrypt (indata, outdata, i, ivec, &num, encordec);

-      for (j = 0; j < i; j++)
-        check += (outdata[j] != out_key[l++]);
-
       i = 0;
     }
   return check;
 }
-
-int
-main ()
-{
-  int main_result;
-
-      main_result = 0;
-      main_result = blowfish_main ();
-
-      printf ("%d\n", main_result);
-
```

```
-      return main_result;
-    }
```

Figure A.3 Modifications apportées à blowfish.c pour réaliser les tests d'*HLSComms*.

JPEG

```
--- a/jpeg/jpeg2bmp.c
+++ b/jpeg/jpeg2bmp.c
@@ -47,7 +47,7 @@ unsigned char JpegFileBuf[JPEG_FILE_SIZE];


 int
-jpeg2bmp_main ()
+jpeg2bmp_main (const unsigned char* in_jpg)
 {
   int ci;
   unsigned char *c;
@@ -58,30 +58,13 @@ jpeg2bmp_main ()
    */
   c = JpegFileBuf;
   for (i = 0; i < JPEGSIZE; i++)
-
-    {
        ci = hana_jpg[i];
+
+    {
+        ci = in_jpg[i];
         *c++ = ci;
     }

   jpeg_read (JpegFileBuf);

-  for (i = 0; i < RGB_NUM; i++)
-    {
-      for (j = 0; j < BMP_OUT_SIZE; j++)
-        {
-          if (OutData_comp_buf[i][j] != hana_bmp[i][j])
-            {
-              main_result++;
-            }
-        }
-    }
-  if (OutData_image_width != out_width)
-    {
```

```
-      main_result++;
-    }
-  if (OutData_image_height != out_length)
-    {
-      main_result++;
-    }
   return (0);
 }

--- a/jpeg/main.c
+++ b/jpeg/main.c
@@ -52,7 +52,7 @@ int
 main ()
 {
   main_result = 0;
-  jpeg2bmp_main ();
+  jpeg2bmp_main (hana_jpg);

   printf ("%d\n", main_result);
```

Figure A.4 Modifications apportées à jpeg.c pour réaliser les tests d'*HLSComms*.

SHA

```
--- a/sha/sha.c
+++ b/sha/sha.c
@@ -24,6 +24,7 @@
 /* NIST's proposed modification to SHA of 7/11/94 may be */
 /* activated by defining USE_MODIFIED_SHA */

+#include "sha.h"

 /* SHA f()-functions */

@@ -77,7 +78,7 @@ local_memcpy (INT32 * s1, const BYTE * s2, int n)
   m = n / 4;
   p1 = (INT32 *) s1;
   p2 = (BYTE *) s2;
-
+  #pragma clang loop interleave_count(SHA_BLOCKSIZE/4)
   while (m-- > 0)
     {
       tmp = 0;
@@ -161,6 +162,7 @@ sha_update (const BYTE * buffer, int count)
```

```
     }
   sha_info_count_lo += (INT32) count << 3;
   sha_info_count_hi += (INT32) count >> 29;
+  #pragma clang loop interleave_count(8192/SHA_BLOCKSIZE)
   while (count >= SHA_BLOCKSIZE)
     {
       local_memcpy (sha_info_data, buffer, SHA_BLOCKSIZE);
@@ -201,7 +203,7 @@ sha_final ()

 /* compute the SHA digest of a FILE stream */
 void
-sha_stream ()
+sha_stream (const unsigned char indata_[VSIZE*BLOCK_SIZE], const int* in_i_)
 {
   int i, j;
   const BYTE *p;
@@ -209,8 +211,8 @@ sha_stream ()
   sha_init ();
   for (j = 0; j < VSIZE; j++)
     {
-      i = in_i[j];
-      p = &indata[j][0];
+      i = in_i_[j];
+      p = &indata_[j*BLOCK_SIZE + 0];
       sha_update (p, i);
     }
   sha_final ();
```

Figure A.5 Modifications apportées à sha.c pour réaliser les tests d'*HLSComms*.